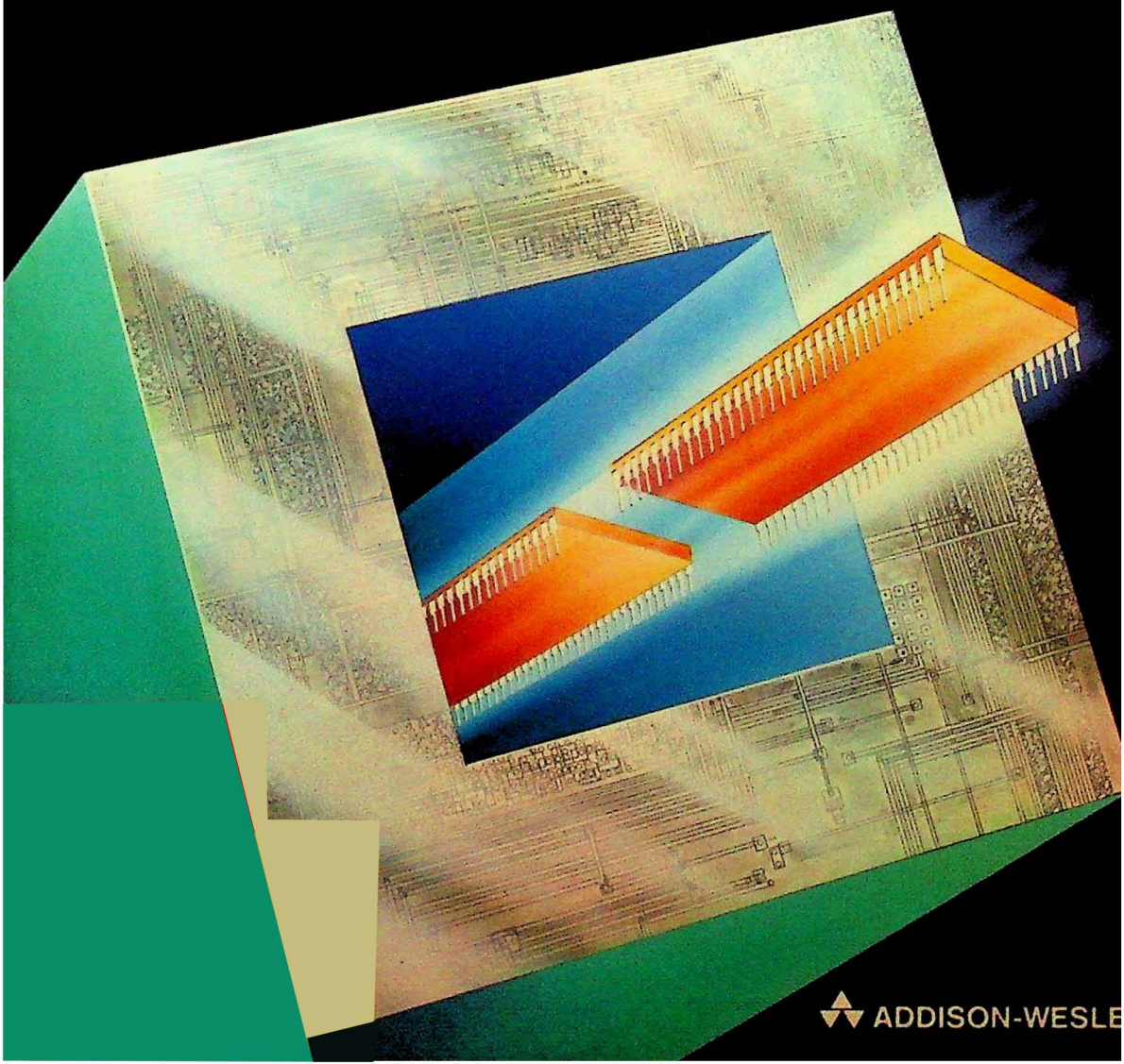
Programmierung des M68000

T. King & B. Knight



ADDISON-WESLE

Programmierung des M68000

Programmierung des **M68000**

T. King & B. Knight

Die englische Originalausgabe erschien unter dem Titel: PROGRAMMING THE M68000, innerhalb der 'Small Computer Series'

© 1983 Addison-Wesley Publishers Limited

© 1986 Addison-Wesley Verlag (Deutschland) GmbH

Übersetzung: Heribert B. Bieling
Umschlagentwurf: BSE Advertising

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten. Kein Teil dieses Buches darf ohne schriftliche Genehmigung des Verlages in irgendeiner Form durch Fotokopie, Mikrofilm oder andere Verfahren reproduziert oder in eine für Maschinen, insbesondere Datenverarbeitungsanlagen, verwendbare Sprache übertragen werden. Auch die Rechte der Wiedergabe durch Vortrag, Funk und Fernsehen sind vorbehalten.

ISBN 3-925118-07-1

Inhaltsverzeichnis

1 Einführung 13
1.1 Die Entwicklung der Mikroprozessoren 14
1.2 Einführung in den 68000 20
1.3 Positionsunabhängiger Code 22
1.4 Eingebaute Hilfsmittel zur Fehlersuche 23
1.5 Unterstützung höherer Programmiersprachen 24
1.6 Unterstützung des Betriebssystems 26
1.7 Einige typische Anwendungen 27
1.8 Die anderen Prozessoren der 68000-Serie 28

2 Syntax des Assemblers und Adressierungsarten 31
2.1 Die Syntax des Assemblers 31
2.2 Assembleranweisungen 33
2.3 Zusammenfassung der Syntax des Assemblers 36
2.4 Ausdrücke 37
2.5 Adressierungsarten 39
2.6 Einteilung der Operandenadressen 49

3 Verschieben und Vergleichen von Daten 51
3.1 Einfaches Verscheiben von Daten 51
3.2 Bedingte Sprünge 53
3.3 Vergleichen von Daten 55
3.4 Noch mehr bedingte Sprünge 57
3.5 Ein einfaches Programm zur Speicherdiagnose 58
3.6 Die Null 60
3.7 Kleine Zahlen 61
3.8 Testen einzelner Bits 61
3.9 Testen von Bedingungen 62
3.10 Schleifen 63
3.11 Einfache Ein- und Ausgabe 65
3.12 Datenausgabe an Peripheriegeräte 68

4 Stacks und Unterprogramme 71
4.1 Unterprogramme 75
4.2 Absolute Sprünge 80
4.3 Operandenadressen 82
4.4 Anforderung von Stackbereichen 84
4.5 Ein Speichertest-Programm 88

5 Arithmetik 91
5.1 Addition 91
5.2 Subtraktion 93
5.3 Negieren 94
5.4 Multiplikation 94
5.5 Austausch von Registern 95
5.6 Multiplikation von Vollworten 95
5.7 Division 100
5.8 Division von Vollworten 101
5.9 Dezimalzahlen 104

6 Logische Operationen 111
6.1 Shiften und Rotieren 113
6.2 Umwandlung einer Hexadezimalzahl in ihre Zeichendarstellung 115
6.3 Operationen auf einzelnen Bits 116
6.4 Ein Speicherverwaltungspaket 118

7 Behandlung von Exceptions 123
7.1 Exceptionvektoren 124
7.2 Benutzer- und Supervisormodus 126
7.3 Abarbeitung von Exceptions 127
7.4 Exception-Handler 128
7.5 Interrupts 129
7.6 Externer Reset 130
7.7 Illegale und nicht implementierte Befehle 130
7.8 Befehle, die einen Trap verursachen können 131
7.9 Privilegverletzung 132
7.10 Tracen 134
7.11 Bus- und Adreßfehler 134
7.12 Mehrfache Exceptions 136
7.13 Ein Unterprogramm zur Ermittlung der Speichergröße 137

8 Ein komplettes Programm: Ein Monitor 139
8.1 Konstantendefinitionen und Exception-Handler 142
8.2 Ein- und Ausgabe 144

8.3 Sprungtabellen 150
8.4 Initialisierung und Abarbeiten von Monitorbefehlen 151
8.5 Die Unterprogramme zur Behandlung der Befehle 153
8.6 Ausgabe und Modifikation von Registern 154
8.7 Der Start eines Benutzer-Programms 158
8.8 Ausgabe und Veränderung von Speicherstellen 160
8.9 Breakpoints 163
8.10 Behandlung von Exceptions 165
8.11 Meldungen und Tabellen 170

Anhang A Der Befehlssatz des 68000 173
Anhang B Beispielprogramme für Sinclair QL, Atari 520ST und Commodore Amiga 183

Glossar 191

Index 195

Vorwort der Autoren

Wir haben dieses Buch so aufgebaut, daß es vom Anfang bis zum Ende durchgelesen werden kann und dabei eine vollständige Einführung in die Assemblerprogrammierung des 68000 bietet. Der fortgeschrittene Leser findet im Anhang eine Zusammenfassung des Befehlssatzes, die kurz die Eigenschaften jedes Befehls schildert und einen Seitenverweis auf die vollständige Beschreibung im Text enthält.

Die Informationen über die Nachfolgemodelle des 68000, den 68010 und den 68020, stammen aus Vorveröffentlichungen der Firma Motorola. Wir möchten Motorola für diese Informationen und auch für die Erlaubnis danken, Teile ihrer Dokumentation zum 68000 verwenden zu dürfen. Die folgende Erklärung wurde auf Wunsch von Motorola aufgenommen.

Motorola übernimmt keinerlei Verantwortung für Fehler in diesem Text und behält sich das Recht vor, Änderungen an den hier beschriebenen Produkten vorzunehmen, wenn diese dem technischen Fortschritt dienen. Motorola übernimmt keinerlei Verantwortung bei der Anwendung und dem Einsatz der hier beschriebenen Produkte. Die Patentrechte von Motorola sind zu wahren. Die Spezifikationen neuer Produkte können jederzeit ohne vorherige Bekanntmachung geändert werden.

Wir möchten uns bei verschiedenen Kollegen an den Universitäten Cambridge und Bath für ihre Hilfe bei diesem Buch bedanken, inbesondere bei Dr. Arthur Norman, der uns seinen Algorithmus zur Division von Vollworten zur Verfügung stellte. Weiterhin möchten wir uns bei Agi Lehar-Graham bedanken, die die Diagramme zeichnete, und bei Jessica King für ihre Hilfe bei der Erstellung des Index.

Tim King
Brian Knight

Vorwort des Übersetzers

In der vorliegenden deutschen Ausgabe dieses Buches sind verschiedene Fachausdrücke eingedeutscht worden, da sie sich mittlerweile in der Fachsprache eingebürgert haben (wie Breakpoint, Interrupt oder Debugger) oder um den Text flüssiger zu gestalten (Exception-Handler, signiert). Dabei entstanden auch einige neue Verben (tracen, debuggen, shiften), die, Konrad Duden möge mir verzeihen, wie deutsche Verben eingesetzt worden sind.

Die in den Beispielen auftretenden Namen und Label wurden fast alle beibehalten, da die Bildung von Namen mnemonischer Wirkung im Deutschen erfahrungsgemäß recht schwierig und im Bereich der Mikroprozessoren einigermaßen unüblich ist.

Das Glossar wurde der deutschen Ausgabe des Buches angefügt, um einige im Bereich der kleineren Mikroprozessoren unübliche Ausdrücke kurz zu erklären. Es erhebt keinerlei Anspruch auf Vollständigkeit, weder in der Auswahl der behandelten Ausdrücke noch in der deren Erläuterung.

An dieser Stelle möchte ich mich besonders bei Frau Doris Antony bedanken, die die Durchsicht und Korrektur der Übersetzung übernommen hat.

Heribert B. Bieling

1
Einführung

Wie der Titel bereits verrät, behandelt dieses Buch den Mikroprozessor 68000 nur aus der Sicht des Programmierers und geht so gut wie gar nicht auf Hardwarefragen ein. Es ist für Leser gedacht, die Zugang zu einem 68000-Rechner besitzen, und beschäftigt sich damit, wie man diesen auf Maschinenebene effektiv programmiert.

Das Buch ist in sich abgeschlossen; es stellt die Maschinenarchitektur und den Befehlssatz in logischer Reihenfolge vor und kann ohne Kenntnis der Motorola-Dokumentation zum 68000 gelesen werden. Diese wird nur benötigt, wenn es um Details (wie zum Beispiel das Bitmuster bestimmter Befehle) geht.

Bei der Besprechung jedes einzelnen Befehls werden alle Besonderheiten seiner Ausführung aufgeführt: sowohl die Fußangeln, die es zu vermeiden gilt, als auch seine wichtigsten Verwendungsmöglichkeiten. Viele dieser Eigenschaften werden leicht überlesen bzw. nicht erkannt, wenn man nur die formale Definition eines Befehls vor sich sieht. In der Praxis kostet es dann Zeit und verursacht Verwirrung, wenn man auf diese ungeahnten Eigenschaften stößt.

Bei der Einführung jedes einzelnen Befehls werden einige ausführliche Beispiele angegeben, um seinen Gebrauch zu veranschaulichen. Diese Programmfragmente sind dazu geeignet, später in größere Programme eingebaut zu werden. Hier werden sie verwendet, um einen kleinen Monitor aufzubauen, welcher einfache Ein- und Ausgabe- sowie Debugging-Möglichkeiten besitzt.

Der Rest dieses ersten Kapitels schildert kurz die Entwicklungsgeschichte der Mikroprozessoren und vergleicht den 68000 mit einigen anderen Prozessoren. Anschließend folgt eine allgemeine Beschreibung des 68000 mit Angabe einiger typischer Anwendungsgebiete. Das zweite Kapitel führt in die Syntax des Assemblers ein, wie sie in den späteren Kapiteln verwendet wird, und erklärt die Adressierungsarten der Befehle.

Die folgenden Kapitel stellen die Befehle des 68000 in zusammengehörenden Gruppen vor. Kapitel 3 beschreibt die verschiedenen Möglichkeiten, Daten zu lesen, zu schreiben und zu vergleichen. Die Konzepte der Stacks und Unterprogramme werden in Kapitel 4 eingeführt, Kapitel 5 befaßt sich mit den Befehlen, die für arithmetische

Operationen benötigt werden. Dazu gehören auch Divisions- und Multiplikationsroutinen für Zahlen, die zu lang sind, um direkt verarbeitet werden zu können. Die logischen Operationen, die auf einzelnen Bits operieren, werden in Kapitel 6 beschrieben. Diese Befehle werden in einem Programmstück zur Speicherplatzzuweisung verwendet. Kapitel 7 behandelt Interrupts und Traps, wobei das Konzept der Interrupt-Programmierung und der Verwendung von Traps für Systemaufrufe, zur Fehlererkennung und in Debuggern vorgestellt werden. Das letzte Kapitel enthält als vollständiges Beispiel einen kleinen Monitor, der Ein- und Ausgabe über ein Terminal ermöglicht und ein bequemes Hilfsmittel zum Austesten von Machinenprogrammen darstellt.

1.1 DIE ENTWICKLUNG DER MIKROPROZESSOREN

Elektronenrechner wurden im Laufe Ihrer Geschichte immer kleiner und kleiner. Die frühesten Maschinen arbeiteten mit Röhren, benötigten viel Platz und verbrauchten Unmengen von Energie. Die Erfindung des Transistors erlaubte es, Platz- und Energiebedarf um ein Vielfaches zu reduzieren. In den 60er Jahren wurde es möglich, integrierte Schaltkreise, die aus einigen Transistoren und anderen Bauteilen bestehen, auf einem einzigen kleinen Silizium-Chip unterzubringen. Dadurch konnte man einen Computer in einem Gehäuse vernünftiger Größe unterbringen. Anfang der 70er Jahre war die Entwicklung der integrierten Schaltkreise soweit gediehen, daß alle Bestandteile des Zentralprozessors eines einfachen Computers auf einem Chip untergebracht werden konnten — der Mikroprozessor war geboren. Seit damals ist die Entwicklung rapide fortgeschritten, und die heutigen Mikroprozessoren können sich mit konventionellen, diskret aufgebauten Computern durchaus messen.

Die ersten Mikroprozessoren, die nennenswerte Anwendung fanden, konnten nur mit 4 Bit langen Daten rechnen. Zu ihnen gehört z.B. der Intel 4040. Sie wurden für einfache Anwendungen wie z.B. Verkaufsautomaten oder Alarmanlagen eingesetzt. Auch fanden sie in einfachen Telespielen Verwendung; aber da sie zu langsam und zu umständlich im Umgang mit sinnvollen Daten waren und auch nur einen geringen Speicherbereich adressieren konnten, war damit ihre Verwendungsfähigkeit schon erschöpft.

Nach Einführung der 8-Bit-Rechner verbreiteten sich die Mikroprozessoren. Die bekanntesten sind der Intel 8080/8085, der Zilog Z80 (der zum 8080 aufwärtskompatibel ist), der 6502 von MOS Technology und Motorolas 6800/6809. Normalerweise haben als »N-Bit-Prozessoren« bezeichnete Prozessoren einige Möglichkeiten zur Verarbeitung von Daten der Länge 2 * N. Die meisten der erwähnten Prozessoren können arithmetische und logische Operationen auf 16-Bit-Worten ausführen.

Einige dieser 8-Bit-Prozessoren wurden sehr billig und somit zum Einbau in andere Geräte verfügbar. So entstanden auch die relativ leistungsfähigen und billigen Homecomputer. Heute finden in fast allen der im Privat- oder Geschäftsbereich verwen-

deten Personal Computer der Z80 oder der 6502 Verwendung.

Während der Entwicklung der 8-Bit-Mikroprozessoren wurden ihre ungünstigen Eigenschaften (z.B. der Bedarf mehrerer Versorgungsspannungen, mehrphasiger Takteingänge oder mehrfach verwendete Daten- und Adressleitungen) beseitigt. Neuere Entwicklungen verwenden nur eine einfache 5-Volt-Stromversorgung, einen einphasigen Takt (oder auch nur einen Quartz zur Kontrolle eines internen Taktgenerators), und jeder Pin des Gehäuses hat nur eine einzige Funktion. Einige Prozessoren (so z.B. der Z80) unterstützen auch den Einsatz dynamischer Halbleiterspeicher, die einen Refresh benötigen.

Eine andere Entwicklung war die Einführung von Ein-Chip-Computern wie dem TMS9940 der Firma Texas Instruments. Diese haben sowohl den Prozessor als auch Fest- (**ROM**) und Datenspeicher (**RAM**) auf einem einzigen Chip. Dadurch können Computer für Spezialanwendungen in ein einziges Gehäuse gepackt werden, was das Zusammenschalten mit anderen Schaltkreisen sehr vereinfacht. Solche Computer werden meist kundenspezifisch für ein bestimmtes Gerät entwickelt und gefertigt.

Softwaremäßig gesehen war die nächste wichtige Entwicklung die der 16- und 32-Bit-Mikroprozessoren Ende der 70er Jahre. Diese ließen den Unterschied zwischen Mini- und Mikrocomputern verschwimmen, da die meisten Minicomputer 16-Bit-Maschinen waren. Unter den ersten dieser Prozessoren waren die TMS9900-Serie von Texas Instruments, Intels 8086, der TMS99000, Zilogs Z8000 und der 68000 von Motorola. Einige zur Zeit noch nicht erhältliche Neuentwicklungen sind der NS16032 von National Semiconductor und die Prozessoren 80286 und iAPX432 von Intel.

Der 68000 hebt sich von seinen Vorgängern dadurch ab, daß er wohl der erste Mikroprozessor ist, dessen Architektur und Befehlssatz denen eines Großrechners gleichen. Er verfügt über einen sehr großen Adreßraum, über die Fähigkeit, 8- , 16- und 32-Bit-Worte zu verarbeiten, über sechzehn 32-Bit-Register, einige Befehle, die das Übersetzen von höheren Programmiersprachen vereinfachen, einen Supervisor-Modus, mit dem man z.B. unberechtigte Zugriffe auf Speicherbereiche oder Ein- und Ausgabeoperationen verhindern kann, sowie Optionen für den Einsatz in einem Multiprozessorsystem.

Zum Vergleich zeigt die folgende Tabelle die wichtigsten Eigenschaften der oben erwähnten Mikroprozessoren. Beim Vergleich der Geschwindigkeiten sollten Sie beachten, daß neuere Versionen eines Prozessors normalerweise mit höheren Taktfrequenzen arbeiten als die ursprünglichen Exemplare. Daher ist die Geschwindigkeit eher ein Maß dafür, wie lange ein Prozessor bereits auf dem Markt ist, und weniger eine Aussage über die Leistungsfähigkeit des Modells.

MOS Technology 6502

Adreßbereich :	64 KByte
schnellster Befehl :	0.5 Mikrosekunden (4 MHz Takt)
allgemeine Register :	1 (8 Bit)
sonstige Register :	2 8-Bit-Indexregister,
	1 8-Bit-Stackpointer
Interruptebenen :	2

Speicherzellen in der untersten Seite ($0000 .. $00FF) können paarweise zur 16-Bit-Indexadressierung verwendet werden. Der Befehlssatz verfügt über eine Anzahl komfortabler Adressierungsarten, allerdings gibt es keine Befehle zum Verarbeiten von Daten mit mehr als 8 Bit Wortbreite.

Zilog Z80

Adreßbereich :	64 KByte
schnellster Befehl :	1 Mikrosekunde (4 MHz Takt)
allgemeine Register :	2 * 7 (8 Bit)
sonstige Register :	2 16-Bit-Indexregister,
	1 16-Bit-Stackpointer
Interruptebenen :	2

Sechs der 8-Bit-Register können paarweise benutzt werden und ergeben so drei 16-Bit-Register. Der Befehlssatz unterstützt 16-Bit-Arithmetik und Blockverschiebe- und Blocksuchbefehle. Die Befehlssätze von 8080/8085 sind eine Untermenge des Z80-Befehlssatzes.

Motorola 6809

Adreßbereich :	64 KByte
schnellster Befehl :	2 Mikrosekunden (2 MHz Takt)
allgemeine Register :	2 (8 Bit)
sonstige Register :	2 16-Bit-Indexregister,
	2 16-Bit-Stackpointer
Interruptebenen :	3

Die zwei 8-Bit-Register können zu einem 16-Bit-Register zusammengefaßt werden. Der Befehlssatz unterstützt teilweise 16-Bit-Arithmetik und eine 8 * 8-Bit-Multiplikation.

Intel 8086

Adreßbereich :	1 MByte
schnellster Befehl :	0.4 Mikrosekunden (5 MHz Takt)
allgemeine Register :	4 (16 Bit)
sonstige Register :	Basis-, Stack- und Indexregister
Interruptebenen :	2

Der Adreßraum ist in 4 Segmente unterteilt. Diese Bereiche, die sich auch überlappen können, sind für Programm, Daten, Stack und sonstige Verwendung gedacht. Die Adressierung geschieht relativ zu sogenannten Segment-Basisregistern; die Basisadresse eines Segments ist stets ein Vielfaches von 16. Der 8086 verfügt über 24 verschiedene Adressierungsmodi, beherrscht 16-Bit-Multiplikation und -Division mit oder ohne Vorzeichen, unterstützt Schleifenkonstrukte durch eigene Befehle und besitzt unteilbare Lese-Schreib-Operationen (wichtig für Multiprozessing). Der 8088 von Intel ist softwarekompatibel zum 8086, hat aber einen nur 8 Bit breiten Datenbus (im Gegensatz zu 16 Bit beim 8086). Er kann daher mit 8-Bit-Bausteinen (Speicher, I/O-Bausteine) zusammengeschaltet werden.

Die TMS9900-Serie

Adreßbereich :	64 KByte (TMS9900)
schnellster Befehl :	2 Mikrosekunden (4 MHz Takt)
allgemeine Register :	16 (im RAM, nicht intern)
sonstige Register :	Workspacepointer
Interruptebenen :	16 (TMS9900,9995), sonst 4

Die Register liegen in dem RAM-Bereich, auf den die Workspacepointer zeigen. Beim TMS9995 sind sie zusätzlich intern gecacht. Für 16-Bit-Multiplikation und -Division stehen Befehle zur Verfügung.

Die TMS9900-Serie besteht aus einer Familie von Prozessoren mit gemeinsamem Befehlssatz:

9900	Grundmodell
9940	Einzel-Chip-Computer mit eingebautem RAM und ROM
9980/81	Prozessor mit 8 Bit breitem Datenbus; kann nur 16 KByte adressieren
9985	Einzel-Chip-Computer mit eingebautem RAM (aber kein ROM)
9995	Register intern gecacht

TMS99000

Adreßbereich :	64 KByte
schnellster Befehl :	0.5 Mikrosekunden (6 MHz Takt)
allgemeine Register :	16 (im RAM, nicht intern)
sonstige Register :	Workspacepointer
Interruptebenen :	16

Der TMS 99000 kann mit Unterstützung eines Zusatzbausteins bis zu 16 MByte segmentierten Speichers adressieren. Er kann 32-Bit-Worte addieren, subtrahieren und shiften, und er besitzt einen Supervisormodus sowie Test-and-Set-Befehle, um mehrere Prozessoren zu synchronisieren. Die Befehlsdekodierung funktioniert derart, daß Befehlscodes, für die keine festeingebauten Befehle existieren, durch vom Benutzer

zu definierenden Mikro-Code, durch Code im Hauptspeicher oder durch einen Zusatzprozessor abgearbeitet werden können.

Zilog Z8000

Adreßbereich :	8 MByte
schnellster Befehl :	0.75 Mikrosekunden (8 MHz Takt)
allgemeine Register :	16 (16 Bit)
sonstige Register :	Refresh-Zähler, Statusbereichzeiger
Interruptebenen :	2

Es existieren sechs Adreßbereiche, von denen jeder bis zu 8 MByte groß sein kann. Es gibt 2 Versionen dieses Prozessors: eine »segmentierte« mit einem Adreßraum von 8 MByte, und eine »unsegmentierte« mit einem Adreßraum von 64 KByte. Die ersten acht Register können als sechzehn 8-Bit-Register behandelt werden; allgemein können die Register als sechzehn 16-Bit-, acht 32-Bit- oder vier 64-Bit-Register gruppiert werden. Multiplikation kann mit 16- oder 32-Bit-Operanden durchgeführt werden, Division sogar mit 32- oder 64-Bit-Operanden. Geshiftet werden kann mit 8-, 16- oder 32-Bit-Registern. Der Z8000 verfügt über einen Supervisormodus, Test-and-Set-Befehle und andere Operationen, die das Zusammenarbeiten mehrerer Prozessoren unterstützen. Er besitzt Blockkopier- und Übersetzungsbefehle. Der Prozessor verfügt über 8 Adressierungsmodi, aber nur über 4 Arten von Prozessortraps.

Motorola 68000

Adreßbereich :	16 MByte
schnellster Befehl :	0.5 Mikrosekunden (8 MHz Takt)
allgemeine Register :	16 (32 Bit)
sonstige Register :	Benutzerstack-Register
Interruptebenen :	7

Intern besitzt der 68000 einen 32 Bit breiten Datenbus, und die meisten Operationen können auf 8-, 16- oder 32-Bit-Werten ausgeführt werden. Von der vollen 32-Bit-Fähigkeit trennt den 68000 nur das Fehlen von 32-Bit-Multiplikation und -Division. Der Adreßraum ist nicht segmentiert, sondern linear adressierbar. Die Register sind in 8 Daten- und 8 Adreßregister unterteilt, wobei einige der Operationen nur auf einem Registertyp ausgeführt werden können. Eines der Adreßregister ist doppelt ausgeführt; welches der beiden Exemplare gerade verfügbar ist, hängt davon ab, ob sich der Prozessor im Benutzer- oder im Supervisormodus befindet. Es existieren 14 Adressierungsmodi, viele verschiedene Arten von Prozessortraps sowie privilegierte Befehle, die nur im Supervisormodus verfügbar sind. Ein Test-and-Set-Befehl ist ebenfalls vorhanden.

National Semiconductor NS16032 (vorläufige Spezifikation)

Adreßbereich :	16 MByte
allgemeine Register :	8 (32 Bit)
sonstige Register :	Zeiger auf Stackrahmen, statische Variablen, Benutzer- und Interruptstack

Der 16032 verfügt über eine 32-Bit-Architektur sowie über verschiedene Fähigkeiten, die zur Unterstützung höherer Programmiersprachen gedacht sind. Es gibt Register zur Adressierung von Stackrahmen und Variablen höherer Programmiersprachen. Des weiteren existieren Adressierungsmodi, welche die Kommunikation zwischen Programm-Modulen mit verschiedenen Datenbereichen erleichtern, des weiteren solche zur Bearbeitung von Arrays von Objekten bis zur Länge von 8 Byte. Außerdem existieren Bitfelder und Gleitkomma-Arithmetik unterstützende Befehle.

Intel 80286

Adreßbereich :	16 MByte
Schnellster Befehl :	0.2 Mikrosekunden (10 MHz Takt)
allgemeine Register :	8 (16 Bit)
Interruptebenen :	2

Der 80286 ist aufwärtskompatibel zum 8086/8088, daher verarbeitet er für diese Prozessoren geschriebene Programme ohne oder mit nur leichten Änderungen. Die Unterschiede liegen in der Geschwindigkeit sowie in der Unterstützung von Mehrbenutzersytemen, welche dieser Prozessor bietet. Speicherverwaltung und -schutz sind bereits fest auf dem Chip integriert, so daß sich eine externe Verwaltungslogik erübrigt. Alle Befehle können nach einer Exception wieder aufgenommen werden. Dadurch läßt sich ein virtueller Speicher von bis zu einem GByte verwalten. Die Hardware ermöglicht es, beim Auftreten von Interrupts den Prozeß schnell und ohne Eingriff des Betriebssystems zu wechseln.

Intel iAPX 432-System (Vorabinformation)

Der Prozessor des iAPX 432-Systems besteht aus zwei Chips: dem Befehlsdekoder 43201 und der Ausführungseinheit 43202. Ein- und Ausgabe werden durch die Interface-Prozessoren 43203 gehandhabt. Daten können bis zu einer Länge von 32 Bit verarbeitet werden, Gleitkommazahlen werden bis zu 80 Bit Länge unterstützt. Aufgrund der Adressierungsmodi können einzelne Datenstrukturen geschützt werden. Bis zu 16 MByte realen Speichers können adressiert werden, wobei die Software bis zu 1024 GByte virtuellen Adreßraum verwalten kann. Zur Unterstützung von Multiprozessorsystemen, Multiprogrammierung und dynamischer Speicherzuweisung sind bereits Funktionen eingebaut. Die Befehlslängen reichen von einigen bis zu mehreren hundert Bit; dabei müssen Befehle nicht (wie bei anderen Prozessoren) an irgendwelchen Wortgrenzen ausgerichtet werden. Eine hohe Systemzuverlässigkeit läßt sich erreichen, indem

man zwei Prozessoren Pin an Pin zusammenschaltet, so daß der eine die Programmausführung des anderen überwachen kann.

1.2 EINFÜHRUNG IN DEN 68000

Als Hintergrund zu den detaillierten Informationen der folgenden Kapitel bietet der Rest dieses Kapitels einen Überblick über den 68000. Einige Befehle werden mit einer kurzen Beschreibung erwähnt; gedacht ist dies für den Leser, der bereits Umgang mit Assemblern anderer Prozessoren hatte. Sollten Sie etwas nicht verstehen, so stören Sie sich nicht daran. In den späteren Kapiteln werden alle Befehle noch ausführlich erklärt.

Der 68000 besitzt zwei verschiedene Arten von Speicher: die internen **Register** (auf dem Chip) und den externen **Hauptspeicher**. Es gibt 17 Register, von denen aber jeweils nur 16 verfügbar sind. Acht davon sind **Datenregister** (D0 bis D7), die anderen acht **Adreßregister** (A0 bis A7). Jedes dieser Register enthält 32 Bit. Bei den meisten Operationen kann ein beliebiges Register verwendet werden; es gibt aber auch Befehle, die nur mit Daten- oder nur mit Adreßregistern ausgeführt werden können. Für Operationen mit **Wort-** (16 Bit) oder **Vollwortoperanden** (32 Bit) kann ein beliebiges Register benutzt werden, ebenso für die indizierte Adressierung (siehe auch Kapitel 2). Mit **Byteoperanden** können nur Datenregister arbeiten, und nur Adreßregister finden als Stackzeiger und Basisregister Verwendung. Das Adreßregister A7 ist doppelt ausgeführt. Welches dieser Exemplare gerade benutzt wird, hängt von dem Prozessorzustand ab (ob er sich gerade im Supervisormodus befindet oder nicht).

Der Hauptspeicher ist byteweise orientiert; von der vorliegenden Rechnerkonfiguration hängt ab, wieviel verfügbar ist. Jedes dieser Bytes besitzt seine **Adresse**, anhand der es eindeutig identifiziert werden kann. Normalerweise (aber nicht immer) besitzt der Speicher die Adressen 0, 1, 2, ..., N – 2, N – 1, wobei N die Gesamtzahl verfügbarer Bytes ist. Der 68000 kann sehr viele Speicherstellen direkt adressieren – über 16 Millionen Byte. Er kann auf Bytes, Worten und Vollworten im Speicher direkt operieren. Ein Wort besteht dabei aus zwei aufeinanderfolgenden Bytes, wobei das erste eine geradzahlige Adresse hat. Analog dazu besteht ein Vollwort aus vier aufeinanderfolgenden Bytes, ebenfalls bei einer geradzahligen Adresse beginnend. Sowohl Worte als auch Vollworte werden über die Adresse ihres ersten Bytes adressiert.

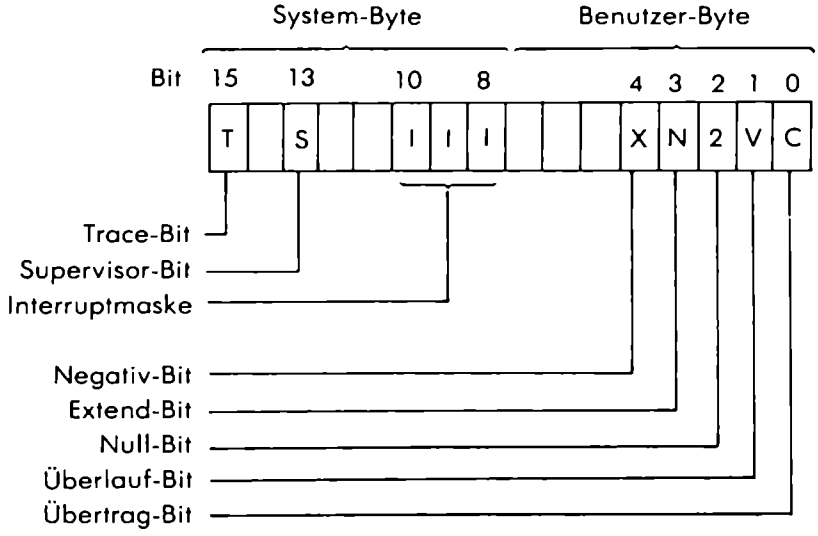
In einem 68000-System kann eine Speicheradresse mit 24 Bit angegeben werden. Wird also für eine Adresse ein Vollwort oder ein Register verwendet, so ergeben sich 8 überflüssige Bits. Daher sind Adressen immer positive Zahlen, und es gibt beim Vergleich zweier Adressen keinerlei Probleme (im Gegensatz zu Computern mit 16 Bit breiten Adressen und ebenso breiten Registern). Andererseits können diese 8 Bit auch verwendet werden, um zusätzliche Information unterzubringen. Diese kann z.B. besagen, auf welchen Datentyp die Adresse zeigt, oder aber auch, daß es sich tatsächlich um

eine Adresse und nicht um einen Wert handelt. An dieser Stelle ist jedoch eine Warnung angebracht: Aus Informationen von Motorola geht hervor, daß bei zukünftigen Modellen der 68000-Serie die vollen 32 Bits für Adressen verwendet werden könnten. Programme, welche die jetzt noch überflüssigen Bits verwenden, werden sich dann nur schwer auf diese neuen Prozessoren übertragen lassen.

Außer den zu bearbeitenden Daten enthält der Hauptspeicher natürlich auch die **Programmbefehle**, welche dem Computer erst sagen, was er zu tun hat. Jeder dieser Befehle belegt ein bis fünf Worte im Speicher, wobei ein Wort für den Befehlscode selbst und die übrigen für die Operanden benötigt werden. Der Befehlscode besagt, was zu tun ist, und er beinhaltet auch die Information darüber, wieviele Operandenworte folgen. Die Operanden geben an, in welchen Registern oder wo im Hauptspeicher sich die zu verarbeitenden Daten befinden, und wo das Ergebnis wieder abzulegen ist.

Normalerweise werden die Befehle in der Reihenfolge abgearbeitet, in der sie im Hauptspeicher vorliegen, etwa so, wie die Noten in einem Musikstück. Ein spezielles Register namens **Programmzähler** enthält stets die Adresse des nächsten auszuführenden Befehls. Einige Befehle ändern diese Ausführungsreihenfolge. Diese sogenannten **Sprung**- oder **Branch**- (Verzweigungs-) Befehle veranlassen den Prozessor, das Programm an einer anderen Adresse weiter abzuarbeiten. So kann der Computer Schleifen ausführen oder in Abhängigkeit von den Daten verschiedene Operationen durchführen.

Ein weiteres Spezialregister ist das **Statusregister**, welches wichtige Informationen über den Systemzustand enthält. Es hat die folgende Form:



Die Bedeutung der Bits im System-Byte wird in Kapitel 7 ausführlich erklärt. Daher hier nur kurz:

Trace-Bit Eins, falls sich der Prozessor im Tracemodus befindet, sonst Null.
Supervisor-Bit Eins, falls sich der Prozessor im Supervisormodus befindet, Null bei Benutzermodus.
Interruptmaske Gibt an, welche der 7 Interruptebenen aktiviert sind.

Das Benutzer-Byte enthält die fünf Bits der **Condition-Codes**. Diese Flags werden von bestimmten Operationen gesetzt (zum Beispiel von Rechen- oder Vergleichsoperationen), und die in ihnen enthaltene Information wird in späteren Operationen ausgenutzt. Die Condition-Codes haben die folgenden Bedeutungen:

Z Das Ergebnis einer Operation war Null.
N Das Ergebnis einer Operation war negativ.
V Bei einer Rechenoperation mit Zweierkomplement-Arithmetik trat ein Überlauf auf (z.B. war das Resultat zu groß, um in den für das Ergebnis vorgesehenen Platz zu passen).
C Ein Übertrag trat auf.
X Erweiterungsflag. Dieses wird bei Operationen mit überlangen Operanden verwendet, z.B. bei der Addition zweier 64-Bit- Zahlen. Es hat hierbei eine ähnliche Wirkung wie das C-Flag, wird aber nicht von so vielen Operationen verändert wie dieses.

Wie die Condition-Codes gesetzt sind, kann mit den Befehlen Bcc, DBcc und Scc abgefragt werden. Diese werden in Kapitel 3 eingeführt.

Die Befehle des 68000 lassen sich in einige wenige Gruppen einteilen, und die folgenden Kapitel befassen sich jeweils mit einer dieser Gruppen. Viele Befehle dienen dazu, Daten von, zwischen und nach Speicherzellen und/oder Registern zu transportieren. Andere führen arithmetische oder logische Operationen (wie Additionen oder Vergleiche) durch. Sprünge und Verzweigungen werden zur Kontrolle und Steuerung der Abarbeitungsreihenfolge verwendet. Wieder andere Befehle können beispielsweise den Prozessor anhalten oder Ein- und Ausgabe-Einheiten steuern.

1.3 POSITIONSUNABHÄNGIGER CODE

Häufig werden Programme so geschrieben, daß für Programmvariablen oder Sprungziele feste Speicheradressen verwendet werden. Derartige Programme müssen an eine bestimmte Stelle im Speicher geladen werden, da sie sonst nicht ablauffähig sind. Dies mag auf einem einfachen Computer oder einem System, das nur ein Programm zur Zeit abarbeiten kann, akzeptabel sein; aber für viele Fälle ist es günstiger, wenn ein Programm an einer beliebigen Stelle gestartet werden kann. Ein solches Programm heißt dann in **positionsunabhängigem** Code geschrieben oder **relokatibel** (verschieblich).

Der Befehlssatz des 68000 erleichtert es, relokatibel zu programmieren, da die meisten Sprungbefehle das Sprungziel nicht **absolut**, sondern **relativ** zur momentanen Position angeben. Dies bedeutet, daß ein Sprungbefehl beispielsweise nicht besagt: »Gehe zur Stelle 5000«, sondern: »Gehe 138 Schritte vorwärts«. Bei dieser Adressierung ist die absolute Lage des Programms egal. Die meisten Mikroprozessoren erlauben derartige Sprünge, jedoch ist der Sprungbereich meist auf 128 Byte nach vorne und nach hinten beschränkt, was häufig nicht ausreicht. Der 68000 erlaubt es, bis zu 32768 Byte weit zu springen, was für jedes Programm ausreichen sollte.

Ein anderer Aspekt der Positionsunabhängigkeit ist die Lage der Programmvariablen. Die reiche Auswahl der Adressierungsmodi (siehe Kapitel 2) erlaubt es, Datenbereiche relativ zu einer in einem Register vorliegenden Adresse zu adressieren, so daß diese an einer beliebigen absoluten Position liegen können.

Ein völlig relokatibles Programm kann auch an einer beliebigen Stelle im Speicher gestartet werden und sich während seiner Ausführung selbst im Speicher verschieben. Ein derartiges Programm muß dann die Datenbereiche relativ zum Programmzähler adressieren. Der 68000 erlaubt derartig adressierte Speicherzellen zu lesen, nicht aber, sie zu beschreiben. Diese Einschränkung ist wohlüberlegt: Sie unterstützt die Programmiertechnik, Programm- und Datenbereiche im Programm strikt zu trennen.

Mit diesen Eigenschaften erlaubt es der 68000, ohne jegliche Tricks Programme zu schreiben, die an beliebigen Stellen geladen und gestartet werden können, ihre Datenbereiche irgendwo anlegen und bei Beibehaltung derselben selbst nach Belieben verschoben werden können.

1.4 EINGEBAUTE HILFSMITTEL ZUR FEHLERSUCHE

Der 68000 verfügt über eine Anzahl von Eigenschaften, welche die Fehlersuche vereinfachen. Darunter fallen sowohl die automatischen Tests auf illegale Operationen als auch vom Benutzer direkt verwendbare Befehle.

Der Prozessor kann beim Auftreten bestimmter Bedingungen einen sogenannten Hardwaretrap auslösen. Dies bedeutet, daß der normale Programmablauf unterbrochen und unter Rettung des Prozessorzustandes (Programmzähler, Status) zu einer bestimmten Adresse gesprungen wird. An dieser Stelle sollte sich dann ein Programmsegment befinden, welches eine angemessene Reaktion auf die vorgefundene Bedingung zeigt. Diese kann beispielsweise aus dem Ausgeben einer Fehlermeldung bestehen, wobei dem Benutzer verschiedene Auswahlmöglichkeiten — wie Wiederaufnahme oder Abbruch des Programms — angeboten werden. Die Wiederaufnahme des Programms kann an der Stelle erfolgen, an der es unterbrochen wurde.

Ein Trap kann u.a. folgende Ursachen haben:

- Zugriff auf ein Wort oder Vollwort mit einer ungeraden Adresse
- Aufruf eines nicht erlaubten oder nicht definierten Befehls
- Versuch des Zugriffs auf nicht existenten Speicher
- Division durch Null
- Interrupt von einem Peripheriegerät

Traps können auch durch verschiedene Operationen ausgelöst werden. Der Befehl TRAPV zum Beispiel verursacht einen Trap, falls bei der letzten Rechenoperation ein Überlauf auftrat. Er kann daher nach jedem Rechenbefehl verwendet werden, wenn dieser Test gewünscht wird. Analog dazu verursacht der Befehl CHK einen Trap, wenn der Wert eines Registers größer ist als ein bestimmter Wert. Dies kann verwendet werden, um sicherzustellen, daß ein Speicherzugriff nur in einem bestimmten Datenbereich erfolgt.

Der Befehl TRAP löst stets einen Trap aus. Er kann an kritischen Stellen des Programms eingefügt werden, um dieses anzuhalten und dann die Registerinhalte und interessante Speicherzellen auszugeben. So kann die Abarbeitung eines Programms abschnittsweise verfolgt werden.

Schließlich besitzt der Prozessor die Möglichkeit, Programme Schritt für Schritt auszuführen. Dies ist ein sehr mächtiges Hilfsmittel, um die genaue Stelle zu finden, an der ein Fehler auftritt. Um dies zu ermöglichen, wird der Prozessor durch Setzen des entsprechenden Bits im Statusregister in den Tracemodus versetzt. In diesem Modus wird nach Ausführung eines jeden Befehls ein Trap ausgelöst. Wird bei diesem Trap ein entsprechendes Programmstück zur Fehlersuche aufgerufen, kann der Anwender die Ausführung jedes einzelnen Befehls beobachten und prüfen. Dieses Verfahren ist auf Prozessoren ohne Tracemodus nur sehr aufwendig zu implementieren. Die Verwendung von Traps zur Fehlersuche wird ausführlich in Kapitel 7 besprochen.

1.5 UNTERSTÜTZUNG HÖHERER PROGRAMMIERSPRACHEN

Der 68000 wurde in dem Bewußtsein entwickelt, daß viele seiner zukünftigen Anwender weniger an der Assemblerprogrammierung als am Einsatz einer der vielen höheren Programmiersprachen wie FORTRAN, Pascal oder C interessiert sind.
Ein Programm in einer dieser Sprachen sieht der Umgangssprache wesentlich ähnlicher als ein Assemblerprogramm. Daher können solche Programme schneller und einfacher geschrieben werden, und auch Fehler sind im allgemeinen einfacher zu entdecken. Derartige Programme können überdies portierbar sein, das heißt, sie können auf jedem beliebigen Rechner laufen, auf dem diese Programmiersprache verfügbar ist. Assemblerprogramme dagegen sind nur auf dem Rechnertyp lauffähig, für den sie geschrieben wurden.

Zur Übersetzung der Befehle einer höheren Programmiersprache in den vom Prozessor verstandenen Maschinencode wird ein Programm namens Compiler verwendet. Da manche Anweisungen mit unnötig vielen Operationen ausgeführt werden, ist der von einem Compiler erzeugte Code meist umfangreicher, langsamer und damit ungünstiger als manuell erstellter. Ein typisches Beispiel hierfür ist ein Befehl, der eine Variable um Eins erhöht. Während der Assemblerprogrammierer diese Variable evtl. in einem Register hält und daher zur Ausführung dieses Befehls nur das Register inkrementiert, holt sich der vom Compiler erzeugte Code die Variable zunächst aus dem Hauptspeicher, addiert dann die Konstante Eins dazu und speichert das Ergebnis wieder ab. Eventuell überprüft der compilierte Code vor dem Abspeichern auch noch, ob bei der Addition kein Überlauf auftrat. Da allerdings Rechen- und Speicherleistung ziemlich preiswert geworden sind, ist es gerechtfertigt, etwas davon zu opfern, um dem Menschen einige Mühen zu ersparen.

Der 68000 besitzt einige Eigenschaften, welche sowohl das Schreiben von Compilern erleichtern als auch diese befähigen, effizienten Code zu erzeugen. Aufgrund der 16 beliebig verwendbaren Register des 68000 können häufig verwendete Zeiger und Variablen im Prozessor gehalten und so unnötige Codestücke, die nur zum Umschaufeln der Daten dienen, eingespart werden. Aufgrund der regulären und in sich konsistenten Struktur der Befehle und der Adressierungsmodi kann der Codegenerierungsteil eines Compilers recht einfach gehalten werden. Die meisten Operationen können auf Byte-, Wort- und Vollwortoperanden ausgeführt werden und erlauben alle Adressierungsmodi. Die Speicherverwaltung bei höheren Programmiersprachen wird durch die Fähigkeit erleichtert, sehr große Speicherbereiche direkt zu adressieren.

Verschiedene Befehle wurden gerade im Hinblick auf höhere Programmiersprachen implementiert. In diesen Sprachen werden meist einzelne Module oder Unterprogramme geschrieben, die zu einem kompletten Programm zusammengestellt werden. Beim Übersetzen eines Moduls weiß der Compiler nicht, von woher es aufgerufen werden wird und welche Register und Speicherbereiche es risikolos verwenden darf. Das Problem der Registerbelegung läßt sich dadurch umgehen, daß man die Registerinhalte bei Einsprung in das Modul rettet und beim Rücksprung dann wiederherstellt. Genau dies ermöglicht der Befehl MOVEM, der eine beliebige Anzahl von Registern in den Speicher schreibt oder von dort wieder zurückholt.

Die Befehle LINK und UNLK erlauben es einem Modul, sich einen eigenen Speicherbereich vom Stack (siehe Kapitel 4) zuzuweisen. Bei Aufruf wird der alte Zeiger auf den Arbeitsbereich weggespeichert und ein neuer auf einen Bereich der angegebenen Größe zurückgegeben. UNLK macht dies rückgängig, gibt also den reservierten Bereich wieder frei und setzt den Zeiger wieder auf den alten Wert zurück.

Die in dem Abschnitt über Fehlersuche vorkommenden Befehle (wie CHK und TRAP) können auch zur Unterstützung des Compilers dienen. Der Compiler kann diese Befehle im generierten Code verwenden, um Überläufe oder unberechtigte Speicherzu-

griffe zu erkennen und zu behandeln. Da dies einzelne Befehle sind, setzen sie die Ausführungsgeschwindigkeit nicht wesentlich herab, und ein Compiler kann sie an jeder notwendigen Stelle einbauen. Solche Stellen wären etwa nach einer Rechenoperation oder nach der Bestimmung eines Index für einen Arrayzugriff. Einem menschlichen Programmierer würde es recht schwer fallen, alle diese Möglichkeiten zu beachten.

1.6 UNTERSTÜTZUNG DES BETRIEBSSYSTEMS

Ein nackter Rechner ist eine ziemlich schwierig zu beherrschende Angelegenheit: er versteht nur die in seiner eigenen, der Maschinensprache geschriebenen, im Speicher bereits vorliegenden Anweisungen. Daher läßt man normalerweise stets ein Programm laufen, das die Handhabung der Maschine vereinfacht. Ein solches Programm nennt man **Betriebssystem** oder, wenn es sehr einfach ist, auch **Monitor**.

Ein typisches Betriebssystem steuert alle Peripheriegeräte eines Rechners, reagiert auf die an der Konsole eingegebenen Befehle und verwaltet die Plattenspeicherung. Es ermöglicht es, mehrere Programme scheinbar gleichzeitig ablaufen zu lassen, indem es in kurzen Zeitabständen zwischen diesen umschaltet; es behandelt auch verschiedene in den laufenden Programmen auftretende Fehler. Dabei kann es Fehlermeldungen ausgeben und dem Benutzer die Möglichkeit geben, sein Programm im Speicher zu untersuchen oder die Registerinhalte zu inspizieren.

Viele der Eigenschaften des 68000 sind für ein Betriebssystem hilfreich oder gar notwendig. So hat das Betriebssystem die Möglichkeit, sich vor Beschädigung durch die anderen Programme zu schützen und deren Ablauf zu steuern. Dies wird durch die beiden verschiedenen Arbeitsmodi des Prozessors erreicht: den Supervisor- und den Benutzermodus. Das Betriebssytem läuft im Supervisormodus und schaltet den Prozessor in den Benutzerstatus, bevor eines der anderen Programme zum Ablauf gelangt. Einige kritische Befehle sind privilegiert und somit nur im Supervisormodus verwendbar. Über eine Ausgangsleitung gibt der Prozessor bei jedem Speicherzugriff an, in welchem Modus er sich befindet, daher können durch zusätzliche Hardware bestimmte Speicherbereiche oder Peripheriegeräte vor der Benutzung durch nichtprivilegierte Programme geschützt werden. So kann das Betriebssystem seinen Programm- und Datenbereich schützen und bewerkstelligen, daß es als einziges auf die Peripheriegeräte zugreifen kann. Dazu muß sichergestellt sein, daß kein Benutzerprogramm in den Supervisormodus gelangen kann, dafür aber Unterprogramme des Betriebssystems aufrufen darf, welche dann im Supervisormodus ablaufen. Die erste Bedingung wird durch die Tatsache gewährleistet, daß die Befehle, mit denen man den Status ändern kann, zu den privilegierten gehören und daher vom Benutzerprogramm nicht verwendet werden können. Die zweite Bedingung wird durch den Befehl TRAP erfüllt (siehe Kapitel 7). Dieser kann einen Sprung ausführen und dabei den Status wechseln.

Der 68000 verfügt über Interrupt- und Trapvektoren. Dadurch kann jedes Peripheriegerät den Prozessor veranlassen, direkt zu einem Programmstück zur Steuerung dieses Geräts zu springen, was zu einer starken Vereinfachung des Betriebssystems führt. Aufgrund der verschiedenen Interruptebenen ist es möglich, die Interrupts verschiedener Geräte ihrer Priorität gemäß abzuarbeiten.

Auch für das Betriebssystem ist der Befehl MOVEM wichtig. Bei Auftreten eines Interrupts oder eines Traps erfolgt unmittelbar ein Sprung. Das angesprungene Programmsegment muß dann die Inhalte aller Register retten, die es selbst verwenden will. Ein weiterer besonderer Befehl ist MOVEP; er vereinfacht den Datenverkehr mit Peripheriegeräten.

Laufen zwei Programme nebenläufig, so ist es mitunter notwendig, einem von ihnen den Zugriff auf ein Betriebsmittel (wie Drucker, Haupt- oder Plattenspeicher) exklusiv zu gestatten. Am einfachsten wird dies durch ein Flag im gemeinsamen Hauptspeicher realisiert, das angibt, ob das Betriebsmittel frei ist. Wird das Betriebsmittel benötigt, wartet das Programm, bis das Flag gesetzt ist, und setzt es dann zurück. Der Vorgang des Testens und Rücksetzens muß jedoch als eine unteilbare Operation erfolgen, da sonst unter Umständen zwei Prozesse das Flag gesetzt vorfinden, es beide zurücksetzen, und dann auch gemeinsam auf das vermeintlich exklusiv verfügbare Betriebsmittel zugreifen. Der Befehl TAS (Test-and-Set) dient dazu, dies zu vermeiden. Er kann auch zur Kommunikation zweier Prozessoren, die einen gemeinsamen Hauptspeicher verwenden, benutzt werden, da der Prozessor während der Ausführung von TAS die Kontrolle über den Speicher behält. Man nennt dies auch einen **Lese-Modifizier-Schreib-Zyklus**.

Der 68000 Prozessor verfügt auch über eine Bus-Steuerungslogik, die es ihm erlaubt, den Prozessorbus mit den Peripheriegeräten zu teilen. Dazu zählen sowohl Hauptspeicher, Plattengeräte und Terminals als auch andere Prozessoren. Einheiten mit Eigenintelligenz können so auch ohne Unterbrechung des Prozessors auf den Speicher zugreifen. Dies wird auch als **direkter Speicherzugriff** oder **DMA** (direct memory access) bezeichnet. Zum Beispiel kann der Prozessor ein Plattenlaufwerk beauftragen, Daten von einer Platte in den Hauptspeicher zu laden. Bei Verwendung der **DMA** erfolgt dies ohne Mitarbeit des Prozessors; dieser kann währenddessen an einem anderen Programm weiterarbeiten und erhält nur bei Beendigung der Übertragung (quasi als Quittung) einen Interrupt.

1.7 EINIGE TYPISCHE ANWENDUNGEN

Der 68000 ist zur Zeit noch zu teuer, um ihn in irgendein Gerät einzubauen, das seiner Geschwindigkeit und Speicherkapazität nicht bedarf. Derartige Anwendungen werden noch einige Zeit die Domäne der 8-Bit-Prozessoren bleiben. Computerterminals, Graphik-Arbeitsplätze, Textverarbeitungseinheiten, medizinische und andere Ge-

räte mit erhöhten Anforderungen gehören eher zu seinen Anwendungsgebieten. Auch ist er Minicomputern jeglicher Größe ein ernsthafter Rivale. Durch seinen großen Adreßraum kann man mit ihm Personal Computer bauen, die Leistungen wie vormals nur Großrechner erbringen. Auch kann er gleichzeitig mehrere Benutzer bedienen, obwohl hierzu Hardware zum Trennen der Speicherbereiche nötig ist.

1.8 DIE ANDEREN PROZESSOREN DER 68000-SERIE

Der 68000 ist nur das erste Modell einer Reihe gleichartiger Prozessoren. Hier soll kurz auf die drei anderen zur Zeit bekannten Modelle eingegangen werden: die Prozessoren 68008, 68010 und 68020.

Der 68008 ist einfach ein 68000 mit einem 8 Bit (statt 16 Bit) breiten Datenbus. Er kann daher einfach an andere 8-Bit-Bausteine (Speicher etc.) angeschlossen werden und vermindert daher den Hardwareaufwand, was allerdings zu Lasten der Geschwindigkeit geht.

Der 68010 ist dem 68000 sehr ähnlich, besitzt jedoch einige Änderungen, um die Unterstützung des Betriebssytems und die Geschwindigkeit zu verbessern. Es gibt ein neues internes Register, das Vektorbasisregister. Dieses enthält die Basisadresse der Interruptvektoren (siehe Kapitel 7). Defaultmäßig ist es auf Null gesetzt (aus Gründen der Kompatibilität zum 68000), kann aber auch jeden anderen Wert annehmen. Dadurch können verschiedene Betriebssystemroutinen ihre Traps ohne großen Aufwand selbst behandeln.

Einige Änderungen betreffen die nach einer Exception auf dem Stack liegende Information. Es ist nunmehr möglich, einen Befehl, der einen Busfehler verursachte, erneut auszuführen. Dies vereinfacht die Implementation eines Systems mit virtuellem Speicher, bei dem jeder Benutzer viel mehr Speicher zur Verfügung zu haben scheint, als physikalisch vorhanden ist. Das Betriebssystem sorgt hierbei dafür, daß sich die gerade benötigten virtuellen Speicherbereiche auch tatsächlich im realen Speicher befinden. Der momentan nicht benötigte Rest wird im Hintergrundspeicher (i.a. Plattenspeicher) gehalten. Die Umsetzung von virtuellen in physikalische Adressen geschieht dadurch, daß der versuchte Zugriff auf real nicht existierende Speicherbereiche einen Busfehler verursacht. Das Betriebssystem holt sich daraufhin den benötigten Bereich in den physikalischen Speicher und nimmt die Bearbeitung bei dem den Fehler verursachenden Befehl wieder auf.

Der 68010 besitzt zwei neue Befehle: MOVEC und MOVES. MOVEC erlaubt den Zugriff auf verschiedene interne Kontrollregister, darunter auch das erwähnte Vektorbasisregister. MOVES erlaubt den Zugriff auf normalerweise nicht zugreifbaren Speicher. Datenzugriffe geschehen normalerweise auf den Benutzer- oder Supervisor-Datenbereich, abhängig vom momentanen Betriebsmodus. Es gibt nun zwei Funktionsregister

(eins für die Quelle, eins für das Ziel einer Operation) mit jeweils drei Bit, die im Supervisormodus mittels MOVEC gesetzt werden können, so daß Programme im Supervisormodus auch auf den Supervisorprogramm-, den Benutzerprogramm- oder den Benutzerdatenspeicher lesend wie schreibend zugreifen können.

Verschiedene Operationen werden auf dem 68010 schneller ausgeführt als auf dem 68000. Darunter sind die 32-Bit arithmetischen- und logischen Operationen sowie die Befehle CLR, Scc und MOVE SR. Des weiteren wurde das Timing bei Busfehlern enschärft.

Der 68020 enthält außer den Zusätzen des 68010 solche zur Erweiterung der 32-Bit-Fähigkeiten. Dazu gehören ein externer 32 Bit breiter Datenbus, 32-Bit-Offsets bei Branchbefehlen sowie 32-Bit-Displacements bei den indizierten Adressierungsmodi. Die Befehle CHK, LINK, UNLK, MUL und DIV verarbeiten jetzt auch 32-Bit-Operanden. Außerdem ist eine weitere Adressierungsart hinzugekommen, die zweifach indirekt indizierte Adressierung ermöglicht.

Der 68020 besitzt einen **Cache-Speicher** für die Befehle. Da diese nun nicht laufend wieder aus dem Hauptspeicher geholt werden müssen, können kurze Schleifen sehr schnell abgearbeitet werden. Außerdem wurde ein Coprozessor-Interface eingebaut, so daß der Befehlssatz durch Hinzunahme anderer Prozessoren (z.B. für Gleitkomma-Arithmetik) erweitert werden kann.

Der 68020 verfügt weiterhin über eine Anzahl neuer Befehle, z.B. um Datenblöcke zwischen den Adreßbereichen zu verschieben, künstliche Ein- und Aussprünge in und aus Prozeduren höherer Programmiersprachen zur realisieren, sowie den Befehl MOVEF zum Verschieben beliebig großer Bitfelder. Die Reihe der Befehle für gepackte Dezimalzahlen wurde um PACK und UNPK erweitert; diese dienen zur Umwandlung von Zeichen in Dezimalzahlen und umgekehrt.

2
Syntax des Assemblers und Adressierungsarten

In diesem Kapitel finden Sie die für die Erläuterung der verschiedenen Maschinenbefehle notwendigen Grundlagen. Erklärt werden dabei die Syntax des Assemblers, d.h wie ein Programm niederzuschreiben ist, und die verschiedenen Adressierungsmodi, also die Art, wie ein Befehl an seine Operanden gelangt.

2.1 DIE SYNTAX DES ASSEMBLERS

Ein Computer selbst versteht grundsätzlich nur **Maschinensprache**. Diese kann als ein Bitmuster bzw. eine Zahlenfolge in seinem Speicher angesehen werden. Derartige Programme sind jedoch für den Menschen kaum lesbar, geschweige denn zu schreiben, daher werden Programme normalerweise in **Assemblersprache** geschrieben. Diese entspricht direkt der Maschinensprache, verwendet aber Mnemonics für die Befehle und Register. Außerdem können im Programm Adressen und andere Werte mit symbolischen Namen bezeichnet werden. Zur Übersetzung der Assemblersprache in Maschinensprache dient ein Programm namens **Assembler**. Die im folgenden beschriebene Assemblersprache entspricht der von Motorola. Sollten Sie den Assembler eines anderen Herstellers verwenden, werden Sie evtl. kleinere Unterschiede feststellen; in diesem Fall sollten Sie das Handbuch zu Rate ziehen.

Ein Programm besteht aus einer Anzahl von Schritten, sogenannten Befehlen, wobei jeder Befehl als eine Zeile der Assemblersprache geschrieben wird. Jeder Befehl besitzt als Namen einen Mnemocode von 3 bis 5 Buchstaben, wobei manche Befehlszeile nur aus diesem Namen besteht. Ein Beispiel dafür ist:

```
NOP
```

Dieser Befehl bewirkt rein gar nichts. (Ein derartiger Befehl ist nicht völlig sinnlos: Bei der Fehlersuche kann er einen unerwünschten Befehl im Programm ersetzen, und er findet Verwendung, wenn sehr kurze Verzögerungen benötigt werden.)

Wie Sie sicher bemerkt haben, steht dieser Name nicht am linken Rand der Zeile. Dies hat einen Grund, den wir später noch ansprechen werden.

In den meisten Fällen ist der Name alleine nicht ausreichend, da zusätzlich noch angegeben werden muß, wo die zu bearbeitenden Daten zu finden sind. Dazu wird ein Operand hinter den Namen des Befehls gesetzt (mit mindestens einem Leerzeichen Abstand), wie z.B.

```
CLR       D3
```

Dieser Befehl setzt die unteren 16 Bit von Datenregister 3 auf Null. Benötigt ein Befehl zwei Operanden, so werden diese durch Kommata (nicht durch Leerzeichen) voneinander getrennt. Der linke Operand gibt dabei meistens an, woher der zu bearbeitende Wert geholt wird, während der rechte besagt, wo der bearbeitete Wert abgelegt werden soll. Man spricht dabei auch von **Quell-** und **Zieloperanden**. Es ist sehr wichtig, daß Sie diese Reihenfolge einhalten, ganz besonders, wenn Sie es bisher anders gewohnt waren. Ein einfaches Beispiel für einen Befehl mit zwei Operanden ist:

```
MOVE      D1,D4
```

Dieser Befehl kopiert die unteren 16 Bit von Datenregister 1 nach Datenregister 4, ohne die oberen Bit zu beeinflussen.

Der 68000 besitzt die nützliche Eigenschaft, daß die meisten seiner Befehle mit Operanden verschiedener Länge arbeiten können: Byte (8 Bit), Wort (16 Bit) und Vollwort (32 Bit). Um die gewünschte Größe anzugeben, wird einfach das Kürzel '.B', '.W' oder '.L' (long word = Vollwort) an den Namen des Befehls angehängt. Fehlt dieses Kürzel, so wird bei den meisten Befehlen automatisch '.W' verwendet. Daher ist der obige Befehl identisch mit:

```
MOVE.W    D1,D4
```

Um alle 32 Bit eines Registers zu kopieren, lautet der Befehl:

```
MOVE.L    D1,D4
```

Analog dazu heißt der Befehl zum Löschen der unteren 8 Bit eines Registers:

```
CLR.B     D3
```

Es ist sinnvoll, dieses Kürzel immer anzugeben (auch das automatisch angenommene '.W'), da ein beim 68000 häufig auftretender Programmierfehler darin besteht, daß versehentlich das falsche Format eines Befehls verwendet wird. Dadurch können die merkwürdigsten Fehler entstehen, die oft schwer zu finden sind. Wenn Sie stets das Kürzel angeben, passiert Ihnen das nicht so leicht.

Der Assembler ignoriert alles, was in einer Zeile hinter den Operanden steht. Daher können in das Programm Kommentare eingefügt werden, um es lesbarer und

verständlicher zu machen. Eine Zeile wird komplett als Kommentar behandelt, wenn sie mit einem Stern beginnt.

```
*     Diese ganze Zeile ist ein Kommentar
      CLR.L    D3          Ein Kommentar hinter einem Befehl
```

Es ist dringend anzuraten, soviel Kommentare wie möglich zu machen. Obwohl dies bei der Programmentwicklung sehr lästig ist, sind diese Programme später sowohl für Außenstehende als auch für den Autor selbst wesentlich leichter lesbar. Dies erleichtert auch Programmänderungen nach einiger Zeit ungemein.

Bei den obigen Beispielen standen die Befehlsnamen nicht am linken Rand. Wenn eine Zeile nicht mit einem Leerzeichen beginnt, wird das erste Wort als **Label** behandelt, d.h. als symbolischer Name der Speicherstelle, ab welcher der in dieser Zeile stehende Befehl abgelegt wird. Der Name eines Labels ist ein beliebiges Wort, bestehend aus Buchstaben und Zahlen, wobei das erste Zeichen ein Buchstabe sein muß. (Die meisten Assembler gestatten noch andere Zeichen in einem Namen. In den in diesem Buch angeführten Beispielen wurde zusätzlich der Unterstrich (' _ ') verwendet, um die Lesbarkeit zu erhöhen.) Der Assembler merkt sich den Namen des Labels mit der zugehörigen Adresse, und das Label kann dann im Programm beliebig als Bezug auf diese Adresse verwendet werden. Besonders nützlich ist dies bei Sprungbefehlen, die das Programm veranlassen, die Ausführung an einer anderen Stelle fortzusetzen.

```
CLRD3    CLR.L    D3          Befehl mit Label
           :
           :
         JMP      CLRD3       Sprung zum Befehl mit Label CLRD3
```

Wenn ihm ein Doppelpunkt folgt, kann ein Label auch vom linken Rand eingerückt werden:

```
CLRD3:  CLR.L    D3
```

Durch den Gebrauch von Labeln ist es nicht nötig, die tatsächliche Adresse des CLR-Befehls zu kennen. Wir brauchen daher auch den JMP-Befehl nicht jedesmal zu ändern, wenn Modifikationen des Programms diese Adresse verändern.

2.2 ASSEMBLERANWEISUNGEN

Außer Befehlen und Kommentaren verarbeitet der Assembler auch **Anweisungen**, die nur ihn selbst betreffen. Sie werden genauso wie Befehle geschrieben, generieren aber (mit Ausnahme von DC und DS) keinen Code. Wir wollen hier nur ein paar der einfachen Anweisungen besprechen, die bei den meisten 68000-Assemblern in der gleichen Form verwendet werden. Die meisten Assembler besitzen zusätzlich noch andere Anweisungen, die z.B. die Ausgabe eines Listings steuern, das Format des erzeugten

Objektmoduls kontrollieren oder Möglichkeiten der bedingten Assemblierung und der Makro-Programmierung unterstützen.

Ein Beispiel einer solchen Anweisung ist EQU. Diese weist einem symbolischen Namen einen konkreten Wert zu (etwa so, wie ein Label der Name einer Adresse ist). Zum Beispiel weist

```
SIZE    EQU     100
```

dem Namen SIZE den Wert 100 zu. Wo auch immer SIZE im Programm verwendet wird, wird der Assembler sich verhalten, als stände stattdessen '100' im Text. Dies ist mehrfach nützlich: Wenn mehrere Programmteile von dieser Größe abhängen, ist es weitaus einfacher, SIZE nur einmal am Anfang zu ändern, als überall im Text die '100' zu suchen und auszutauschen. Indem für Zahlen Mnemonics verwendet werden, kann die Lesbarkeit eines Programmes verbessert werden.

Der Speicher des 68000 kann als Feld von 8 Bit großen **Bytes** verstanden werden, das mit Null beginnend aufwärts indiziert ist. Die Nummer eines Bytes im Speicher wird dabei als Adresse bezeichnet. Es gibt zwei Anweisungen, die dazu dienen, die Adresse eines Programmstücks festzulegen: ORG und RORG. ORG spezifiziert die Anfangsadresse eines Programms (normalerweise die Adresse des ersten Befehls). Die Befehlsfolge

```
        ORG     1024
START   CLR.L   D3
          :
          :
```

veranlaßt den Assembler, Code zu erzeugen, der bei Adresse 1024 beginnend abgelegt wird. Das Label START hat daher den Wert 1024. Der assemblierte Code wird mit dieser Adresse markiert, so daß er später an die richtige Stelle geladen werden kann. Mit ORG beginnender Code wird, da seine Adresse festliegt, auch als **absoluter** Code bezeichnet, seine Label als absolute Symbole. Ein ORG enthaltendes Programm kann, da es explizite Verweise auf absolute Adressen enthält, normalerweise nicht an einer beliebigen Adresse gestartet werden. ORG besitzt noch eine weitere Form: ORG.L. Diese beeinflußt die Art und Weise, wie bei der Assemblierung absolute Adressen umgesetzt werden (dazu später mehr).

In einem Assembler-Programm ist es oft angenehmer, Zahlen in hexadezimaler (Basis 16) anstatt in dezimaler Notation (Basis 10) angeben zu können. Die in hexadezimaler Schreibweise (kurz 'hex') verwendeten Ziffern sind 0 bis 9, gefolgt von A bis F zur Darstellung der Werte 10 bis 15. Beginnen Zahlen mit einem Dollarzeichen ('$'), so behandelt der Assembler sie als Hex-Zahlen.

```
ORG     $400
```

bedeutet daher das gleiche wie ORG 1024 (= 4×256 + 0×16 + 0×1). Auch in diesem Buch wird '$' zur Charakterisierung von Hex-Zahlen verwendet.

Die zu ORG entgegengesetzte Anweisung lautet RORG und bewirkt, daß das Programm als **relokatibel** angesehen wird, daß es also an einer beliebigen Stelle gestartet werden kann. RORG besitzt wie ORG ein Argument, das aber normalerweise Null sein sollte. Wenn wir das obige Programmstück zu

```
        RORG    0
START   CLR.L   D3
         :
         :
```

abändern, ist der Wert von START dem Assembler unbekannt. START erhält daher den Wert Null (das ist der Abstand bzw. Offset vom Anfang des Abschnitts) und es wird vermerkt, daß es sich um eine **relokatible** Adresse handelt. Label wie START sind Beispiele für relokatible Symbole, deren Wert unbekannt ist, bis sie an eine Adresse im Speicher geladen worden sind. Der Assembler erzeugt beim Auftreten relokatibler Symbole positionsunabhängigen Code. Daher wird bei

```
        RORG    0
START   CLR.L   D3
         :
         :
        JMP     START
```

der Sprungbefehl codiert als: »Springe zu dem Befehl, der X Byte vorher liegt«, wobei X vom Assembler berechnet wird. Wird ein relokatibler Wert derart verwendet, daß die Erzeugung positionsunabhängigen Codes nicht möglich ist, so fügt der Assembler dem Code eine Liste all der Worte hinzu, in die nach dem Laden des Codes noch ein Wert eingetragen werden muß (da diese Werte vorher nicht bekannt sind). Diese Liste wird als **Relokationsliste** bezeichnet.

Zur Reservierung und Initialisierung von Speicherplatz gibt es zwei Befehle: DS und DC. Die Anweisung DS (define storage) wird zur Reservierung von Speicherbereichen verwendet. Die Größe dieser Bereiche wird durch das angehängte Kürzel angegeben, und ein Operand spezifiziert, wieviele dieser Bereiche zu reservieren sind.

```
BUFFER  DS.B    80          Reserviert 80 Byte
        DS.W    $20         Reserviert 32 Worte
        DS      $20         Reserviert ebenfalls 32 Worte
        DS.L    3           Reserviert 3 Vollworte
```

Der reservierte Speicherbereich wird nicht mit einem bestimmten Wert vorbelegt. Wird nicht das Kürzel '.B' an die Anweisung gehängt, so wird der reservierte Bereich an einer Wortgrenze ausgerichtet. 'DS.W 0' kann daher zur Ausrichtung an einer Wortgrenze verwendet werden. Steht vor der DS-Anweisung ein Label, so bezeichnet dies die erste reservierte Adresse (nach der Ausrichtung).

Die Anweisung DC (define constant) wird verwendet, um bei der Assemblierung bestimmte Speicherzellen mit festen Werten zu belegen. Sie verwendet auch die drei Kürzel und erhält einen oder mehrere durch Kommata getrennte Operanden. Wie bei DC erfolgt dabei Ausrichtung an einer Wortgrenze, wenn nicht das Kürzel '.B' angehängt ist. Als Operanden finden Zahlen, Ausdrücke oder in Hochkommata eingeschlossene Zeichenketten Verwendung. Eine Zeichenkette wird dabei nicht als Zeichenkonstante (siehe auch »Ausdrücke« weiter unten) behandelt, sondern für jedes Zeichen wird ein Byte erzeugt. Bei Verwendung von DC.W oder DC.L wird das letzte Wort oder Vollwort (falls nötig) rechts mit Nullen aufgefüllt.

```
MESSAGE  DC.B    'Hello'    5 Byte mit den
*                           Codes für 'H','e' ...
         DC.L    'Hello'    Hierbei werden 8 Byte erzeugt, wobei
*                           die letzten drei den Wert Null
*                           erhalten
         DC.W    10,20,30   3 Worte werden generiert
         DC.L    $FF,99     2 Vollworte werden erzeugt
```

Die Anweisung END wird verwendet, um das Ende eines Programms zu kennzeichen. Die letzte Programmzeile sieht also wie folgt aus:

```
         END
```

Zusammenfassung der Assembleranweisungen

Anweisung			*Funktion*
[Label]	DC.s	exp,exp..	Bestimmt Werte der Größe s
[Label]	DS.s	n	Reserviert n Bereiche der Größe s
	END		Ende des Quellprogramms
Symbol	EQU	Wert	Setzt das Symbol auf den Wert
	ORG	Adresse	Setzt die Anfangsadresse eines absoluten Abschnitts
	RORG	Adresse	Setzt die Anfangsadresse eines relokatiblen Abschnitts

2.3 ZUSAMMENFASSUNG DER SYNTAX DES ASSEMBLERS

Es gibt hauptsächlich drei Arten von Assemblerzeilen: Kommentarzeilen, Befehlszeilen und Anweisungszeilen. Eine Kommentarzeile beginnt mit einem Stern; im Rest der Zeile können beliebige Zeichen stehen.

```
*          Dies ist eine Kommentarzeile
```

Eine Befehlszeile hat im allgemeinen folgende Form:

```
Label   Befehl Operand(en) Kommentar
```

Jedes Feld ist von den anderen durch mindestens ein Leerzeichen getrennt, und in den einzelnen Feldern dürfen keine Leerzeichen stehen (ausgenommen natürlich im Kommentar und bei in Hochkommata eingeschlossenen Zeichenketten). Label und Kommentar sind optional. Das Befehlsfeld beinhaltet den Namen des Befehls, eventuell gefolgt von einem Längenkürzel ('.B', '.W', '.L' oder '.S'). Die Anzahl der Operanden ist durch den Befehlscode festgelegt; benötigt ein Befehl keine Operanden, so behandelt der Assembler alles, was hinter dem Befehlscode steht, als Kommentar. Zwei Operanden müssen durch Kommata (keine Leerzeichen) getrennt werden.

Eine Anweisungszeile hat im allgemeinen folgende Gestalt:

```
Label     Anweisung  Argument(e) Kommentar
```

Ein Labelfeld ist nicht bei allen Anweisungen erlaubt; bei einigen anderen muß dagegen eins vorhanden sein. Zwei oder mehr Operanden müssen durch Kommata getrennt werden.

2.4 AUSDRÜCKE

Wie wir oben gesehen haben, kann eine Zahl stets durch ein ihr entsprechendes Symbol ersetzt werden. Weiterhin kann eine Zahl durch einen arithmetischen Ausdruck ersetzt werden, der Symbole und Zahlen enthält. Es gibt eine Anzahl verwendbarer arithmetischer Operationen wie +, −, * (Multiplikation) und / (Division). Mögliche Ausdrücke sind zum Beispiel:

```
DAYHRS    EQU       24          Stunden eines Tages
DAYMIN    EQU       DAYHRS*60   Minuten eines Tages
DAYSEC    EQU       DAYMIN*60   Sekunden eines Tages
```

Alle Ausdrücke werden nach den üblichen Rechenregeln ausgewertet, und die Ergebnisse sind stets ganze Zahlen. Besonders bedeutend ist dies bei der Division, deren Ergebnis abgeschnitten wird. 7/3 beispielsweise ergibt also 2.

Zahlen können in dezimaler oder in hexadezimaler Darstellung verwendet werden. Eine weitere Möglichkeit ist die der Verwendung von Zeichenkonstanten. Dies sind bis zu vier in Hochkommata eingeschlossene Zeichen, die zu einem Langwort assembliert werden. Dieses enthält die den Zeichen entsprechenden Codes in den äußersten rechten Byte; evtl. links leerbleibende Byte werden mit Nullen aufgefüllt. Zeichenkonstanten werden meistens für einzelne Zeichen verwendet und sollten an stelle von Zahlen verwendet werden, wenn dadurch die Lesbarkeit des Programms zunimmt. Beispiele für Zeichenkonstanten sind:

```
CHARZ     EQU       'Z'         Der Code für den Buchstaben Z
CASEDIFF  EQU       'A'-'a'     Differenz zwischen den Codes für
*                                 Groß- und Kleinbuchstaben
```

Wie wir gesehen hatten, gibt es zwei Arten Symbole: absolute und relokatible. Bei absoluten Symbolen gibt es bei Berechnungen keinerlei Probleme, da sie ja für feste Zahlen stehen. Anders verhält es sich bei relokatiblen Symbolen. Hier gibt es einige Einschränkungen bezüglich dessen, was erlaubt ist, und was vernünftige Ergebnisse liefert. Die Grundregel dabei lautet, daß das Ergebnis einer Berechnung entweder ein absoluter Wert ist, oder aber es ist auf die gleiche Weise relokatibel wie die im Ausdruck stehenden Operanden. Daher sind Multiplikation und Division mit relokatiblen Operanden genauso wenig erlaubt wie die Addition zweier relokatibler Werte. Eine Konstante kann hingegen zu einem relokatiblen Wert addiert oder von einem solchen abgezogen werden, wobei das Ergebnis wieder ein relokatibler Wert ist (eine andere Adresse im gleichen relokatiblen Abschnitt). Man könnte also ein Programmstück

```
        RORG    0           relokatibler Abschnitt
START   CLR.L   D3          Dieser Befehl ist zwei Byte lang
        CLR.L   D4
         :
         :
        JMP     START+2     Sprung zum zweiten CLR.L
```

schreiben, wobei jedoch die Verwendung eines weiteren Labels für das gewünschte Sprungziel wesentlich angebrachter wäre.

Relokatible Werte dürfen nicht von absoluten, wohl aber voneinander subtrahiert werden. Dabei ergibt sich ein absoluter Wert, nämlich die Distanz zweier Adressen im Programm, die unabhängig von der absoluten Lage des Programms im Speicher ist. In dem Programm

```
        RORG    0                   relokatibler Abschnitt
PSTART  MOVE.L  PEND-PSTART,D0      Setzt D0 auf die Länge des
*                                   Programms
         :
         :
PEND
```

wird durch den ersten Befehl D0 mit der Länge des Programms (in Byte) geladen. Hierbei haben wir ein alleinstehendes Label verwendet, das für die erste freie Speicherstelle hinter dem Programm steht.

Die aufgeführten Regeln lassen sich zu einer Tabelle zusammenfassen:

Ausdruck	*Ergebnis*
relokatibel * relokatibel	(nicht erlaubt)
relokatibel * absolut	(nicht erlaubt)
relokatibel / relokatibel	(nicht erlaubt)
relokatibel / absolut	(nicht erlaubt)
relokatibel + relokatibel	(nicht erlaubt)
relokatibel + absolut	relokatibel
relokatibel − relokatibel	absolut
relokatibel − absolut	relokatibel
absolut − relokatibel	(nicht erlaubt)

2.5 ADRESSIERUNGSARTEN

Die meisten Befehle des 68000 können ihre Operanden auf viele Arten erhalten. Sie können in Registern vorliegen, in auf verschiedene Weise zu adressierenden Speicherstellen oder auch im Befehl selbst enthalten sein. Da dies für alle Befehle gilt, können die Adressierungsarten ohne Berücksichtigung einzelner Befehle beschrieben werden. Bei der durch Auswertung des Adressierungsmodus erhaltenen **Operandenadresse** spricht man auch von der 'effective address', der tatsächlichen Adresse des Operanden.

Registeradressierung

Operandenwerte können in einem der Daten- oder Adreßregister vorliegen. Als Registernamen werden dabei Dn bzw. An verwendet, wobei n eine Ziffer zwischen 0 und 7 ist. Der Befehl

```
MOVE.L   A7,D5
```

beispielsweise kopiert alle 32 Bit vom Adreßregister 7 in das Datenregister 5. Wird als Länge '.W' (Wort) spezifiziert, werden nur die hinteren 16 Bit des Registers gelesen bzw. verändert. Die Längenangabe '.B' (Byte) darf im Umgang mit Adreßregistern nicht verwendet werden; bei Datenregistern werden dann nur die letzten 8 Bit beeinflußt.

Absolute Adressierung

Ein Operand im Hauptspeicher kann anhand der absoluten Adresse seines ersten (höchstwertigen) Bytes spezifiziert werden. Der Operand wird dann einfach als

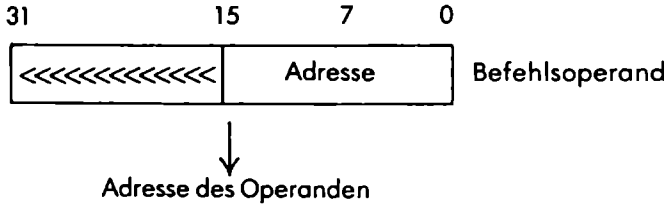
Zahl, Label oder Symbol, das für diese Adresse steht, dargestellt. Um das Byte an der Adresse 1000 (hex) zu löschen, können wir den folgenden Befehl verwenden:

```
CLR.B    $1000
```

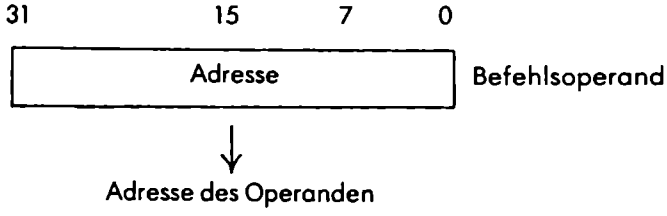
Zu dieser Adressierungsart gibt es zwei unterschiedliche Formate, da die absolute Adresse als Operand in einem Wort (16 Bit) oder in einem Vollwort (32 Bit) stehen kann. Bei der kurzen Form wird die Adresse zunächst auf 32 Bit **'sign-extended'**, d.h. das höchstwertige (am weitesten links stehende) der 16 Bit der Adresse wird in die 16 höchstwertigen (vorher leeren) Bit kopiert. Daher können mit dem kurzen Format nur zwei kleine Bereiche im Hauptspeicher adressiert werden: die unteren und die oberen 32 KByte. Der Bereich dazwischen ist mit der kurzen Form nicht adressierbar. Bei Rückwärts-Verweisen kann der Assembler stets die korrekte Form der Adressierung wählen, da ihm die bezogene Adresse bereits bekannt ist. Bei Vorwärtsreferenzen muß dem Assembler vorher angegeben werden, welche Form er wählen soll. Dies geschieht durch Anhängen eines Längenkürzels an die Anweisung ORG. Ohne Kürzel wird bei Vorwärtsverweisen stets die kurze Form gewählt, bei Angabe von ORG.L hingegen die lange Form. Ein absolutes Programm, das mehr als 32 KByte (= $8000) Speicher benötigt, muß daher mit ORG.L assembliert werden, damit der Assembler die Adressierungsart korrekt wählen kann.

Diese Adreßrechnungen finden Sie nun als Diagramm. '<<<' bedeutet dabei **Sign-Extension.**

Absolute Adressierung (Wortformat)



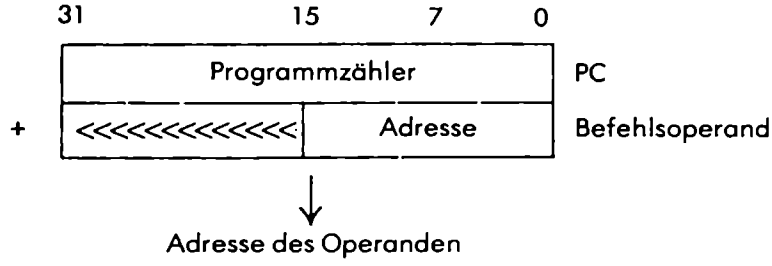
Absolute Adressierung (Vollwortformat)



Relative Adressierung

Es gibt zwei Adressierungsarten, die eine Adressierung relativ zum momentanen Wert des **Programmzählers** (PC, program counter) gestatten. Sie werden meistens bei Sprüngen innerhalb von positionsunabhängigem Code verwendet, können jedoch auch zum Adressieren von im Programm eingebauten Konstanten verwendet werden. Eine derart adressierte Stelle kann nicht beschrieben werden; dadurch wird die Erzeugung **reinen Codes** erzwungen. Derartiger Code verändert sich beim Ablauf nicht selbst und kann somit immer wieder mit stets gleicher Wirkung ablaufen. Er kann auch als Teil mehrerer, gleichzeitig ablaufender Programme verwendet werden. Reiner Code wird auch als **'re-entrant'** bezeichnet, da er jederzeit wiederverwendbar ist. Einen etwas schwächeren Grad von »**Reinheit**« besitzt Code, der als **'serially reusable'** (später wiederverwendbar) bezeichnet wird. Dieser kann nach komplettem Ablauf wieder aufgerufen werden, ist aber während des Ablaufs nicht immer in einem bestimmten Zustand (etwa durch Verwendung von im Code liegenden Datenbereichen).

Bei der einfacheren dieser beiden Adressierungsarten ergibt sich die Operandenadresse einfach als Summe des momentanen Wertes des Programmzählers und eines 16 Bit langen **Displacements** (Versatz). Während der Ausführung eines Befehls ist der Programmzähler um zwei größer als die Adresse des Befehls. Das Displacement wird als signierte, d.h. mit Vorzeichen versehene Zahl betrachtet, so daß Adressen im Bereich Befehl-32766 bis Befehl + 32769 dargestellt werden können.



Diese Adressierungsweise wird vom Assembler verwendet, wenn die Adresse des Operanden als Offset zur momentanen Position dargestellt wird. Dabei dient das Zeichen '*' zum Bezug auf die augenblickliche Position. Daher kann man zwar Befehle wie

```
        JMP     *+10        Sprung zum Befehl 10 Byte weiter
*                           vorn
```

schreiben, sollte dies aber tunlichst unterlassen, da der Offset dabei von Hand ausgerechnet werden muß. Außerdem muß er neu bestimmt werden, wenn man im Zuge der Programmentwicklung weitere Befehle zwischen Sprung und Sprungziel einfügt. Die Verwendung von Labeln ist wesentlich einfacher und sicherer.

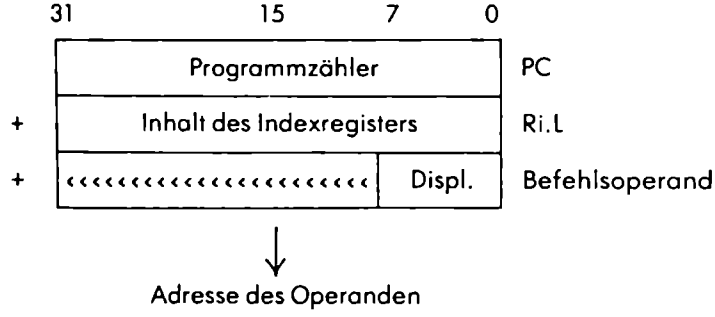
Diese Adressierungsart wird vom Assembler auch automatisch gewählt, wenn in einem relokatiblen Abschnitt ein Verweis auf ein relokatibles Symbol desselben Abschnitts auftritt. Bei der Befehlsfolge

```
          RORG      0
START     CLR.L     D3
            :
            :
          JMP       START
```

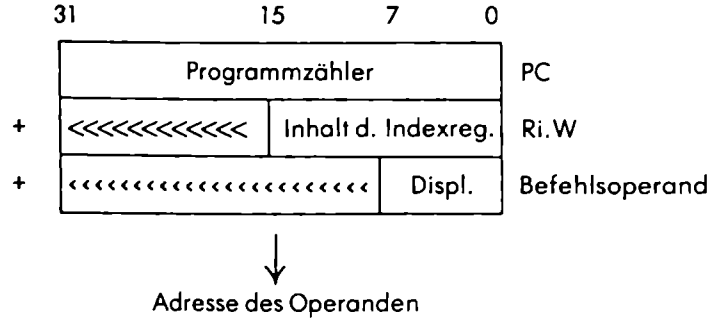
verwendet der Assembler die relative Adressierung zur Angabe des Sprungziels beim Sprungbefehl.

Die andere relative Adressierungsart arbeitet ähnlich, jedoch wird zusätzlich zum Displacement noch der Inhalt eines der Register addiert. Dieses Register wird dann auch als **Indexregister** bezeichnet; geeignet dafür ist jedes Register. Ob das ganze oder nur die rechte Hälfte des Registers verwendet wird, hängt von dem an den Registernamen angehängten Kürzel ab. Bei '.W' (Standard) werden die vorderen 16 Bit des Registers durch Sign-Extension aufgefüllt, bei '.L' werden die ganzen 32 Bit ohne Modifikation verwendet. Der Wert für das signierte Displacement kann bei dieser Adressierungsart nur 8 Bit groß sein. Dadurch ergibt sich ein Bereich von − 128 bis + 127.

Langwortformat



Wortformat



Diese Adressierungsart wird meistens dann benutzt, wenn eines von mehreren Sprungzielen in einer vorherigen Berechnung bestimmt wurde. Sie wird vom Assembler automatisch verwendet, wenn der Operand ein relokatibles Symbol ist, gefolgt vom geklammerten Namen eines Daten- oder Adreßregisters. Ein Beispiel:

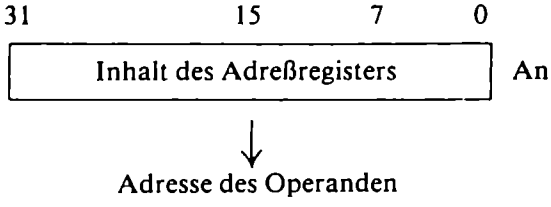
```
            RORG 0
             :                  Codestücke, die eine anzuspringende
             :                  Routine bestimmen und entsprechend
             :                  0, 4, 8 oder 12 in Register A0
             :                  ablegen.
             :                  (Ein Sprungbefehl benötigt 4 Byte)
*
            JMP       JTABLE(A0) Sprung zur gewünschten Routine
*
JTABLE      JMP       ROUTINEA   Wird aufgerufen, wenn A0 0 enthält
            JMP       ROUTINEB   Wird aufgerufen, wenn A0 4 enthält
            JMP       ROUTINEC   Wird aufgerufen, wenn A0 8 enthält
            JMP       ROUTINED   Wird aufgerufen, wenn A0 12 enthält
```

Auch diese Adressierungsart hat zwei verschiedene Formen, je nachdem, ob 16 oder 32 Bit des Indexregisters verwendet werden sollen.

```
            JMP       JTABLE(A0.W)   bewirkt das gleiche wie in dem
*                                    obigen Beispiel. Nur die rechten
*                                    (niederwertigen) 16 Bit von A0
*                                    werden verwendet und als signierte
*                                    16-Bit-Zahl behandelt.
            JMP       JTABLE(A0.L)   verwendet alle 32 Bit von A0
```

Indirekte Adressierung über Adreßregister

Fünf weitere Adressierungsarten basieren darauf, daß die Adresse des Operanden in einem der Adreßregister vorliegt. Da der Inhalt des Registers nicht selbst den Operanden darstellt, sondern nur auf diesen zeigt, nennt man dies **indirekte** Adressierung.



Bei der einfachsten dieser Adressierungsarten wird nur der Name des Adreßregisters in Klammern angegeben, zum Beispiel:

```
CLR.B     (A2)
```

Dieser Befehl löscht das Byte, dessen Adresse in A2 steht.

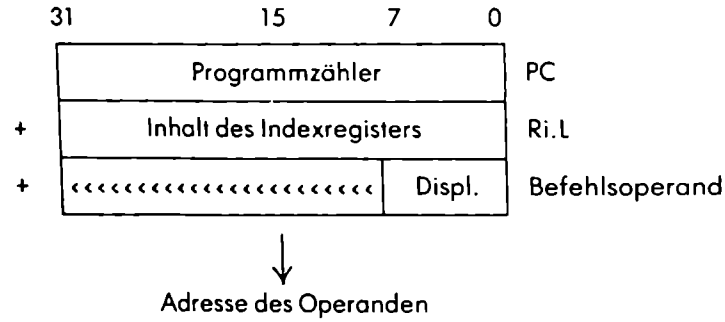
Diese Adressierungsart wird vom Assembler auch automatisch gewählt, wenn in einem relokatiblen Abschnitt ein Verweis auf ein relokatibles Symbol desselben Abschnitts auftritt. Bei der Befehlsfolge

```
        RORG    0
START   CLR.L   D3
          :
          :
        JMP     START
```

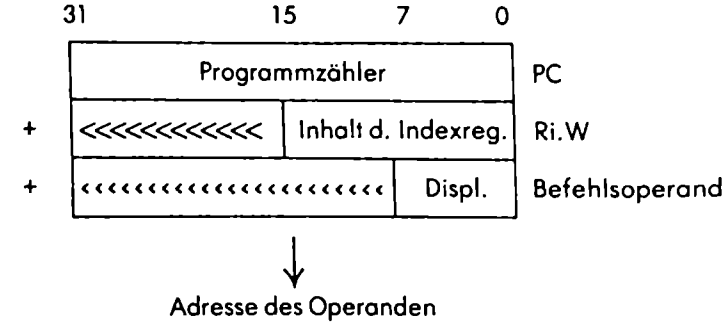
verwendet der Assembler die relative Adressierung zur Angabe des Sprungziels beim Sprungbefehl.

Die andere relative Adressierungsart arbeitet ähnlich, jedoch wird zusätzlich zum Displacement noch der Inhalt eines der Register addiert. Dieses Register wird dann auch als **Indexregister** bezeichnet; geeignet dafür ist jedes Register. Ob das ganze oder nur die rechte Hälfte des Registers verwendet wird, hängt von dem an den Registernamen angehängten Kürzel ab. Bei '.W' (Standard) werden die vorderen 16 Bit des Registers durch Sign-Extension aufgefüllt, bei '.L' werden die ganzen 32 Bit ohne Modifikation verwendet. Der Wert für das signierte Displacement kann bei dieser Adressierungsart nur 8 Bit groß sein. Dadurch ergibt sich ein Bereich von − 128 bis + 127.

Langwortformat



Wortformat



Diese Adressierungsart wird meistens dann benutzt, wenn eines von mehreren Sprungzielen in einer vorherigen Berechnung bestimmt wurde. Sie wird vom Assembler automatisch verwendet, wenn der Operand ein relokatibles Symbol ist, gefolgt vom geklammerten Namen eines Daten- oder Adreßregisters. Ein Beispiel:

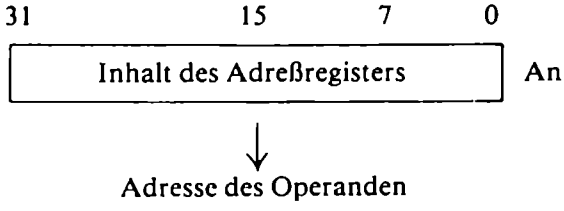
```
          RORG 0
           :                    Codestücke, die eine anzuspringende
           :                    Routine bestimmen und entsprechend
           :                    0, 4, 8 oder 12 in Register A0
           :                    ablegen.
           :                    (Ein Sprungbefehl benötigt 4 Byte)
*
          JMP       JTABLE(A0)  Sprung zur gewünschten Routine
*
JTABLE    JMP       ROUTINEA    Wird aufgerufen, wenn A0 0 enthält
          JMP       ROUTINEB    Wird aufgerufen, wenn A0 4 enthält
          JMP       ROUTINEC    Wird aufgerufen, wenn A0 8 enthält
          JMP       ROUTINED    Wird aufgerufen, wenn A0 12 enthält
```

Auch diese Adressierungsart hat zwei verschiedene Formen, je nachdem, ob 16 oder 32 Bit des Indexregisters verwendet werden sollen.

```
          JMP       JTABLE(A0.W)   bewirkt das gleiche wie in dem
*                                  obigen Beispiel. Nur die rechten
*                                  (niederwertigen) 16 Bit von A0
*                                  werden verwendet und als signierte
*                                  16-Bit-Zahl behandelt.
          JMP       JTABLE(A0.L)   verwendet alle 32 Bit von A0
```

Indirekte Adressierung über Adreßregister

Fünf weitere Adressierungsarten basieren darauf, daß die Adresse des Operanden in einem der Adreßregister vorliegt. Da der Inhalt des Registers nicht selbst den Operanden darstellt, sondern nur auf diesen zeigt, nennt man dies **indirekte** Adressierung.



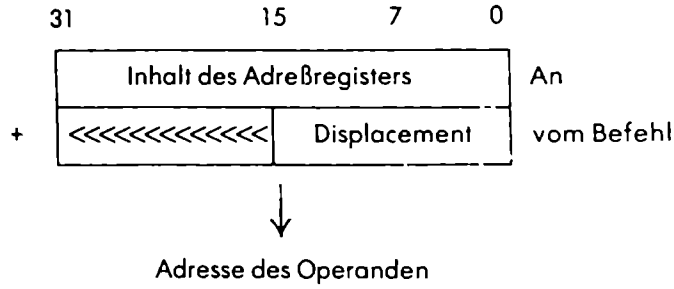
Bei der einfachsten dieser Adressierungsarten wird nur der Name des Adreßregisters in Klammern angegeben, zum Beispiel:

```
CLR.B     (A2)
```

Dieser Befehl löscht das Byte, dessen Adresse in A2 steht.

Indirekte Adressierung mit Displacement

Die in einem Adreßregister stehende Adresse kann durch Addition eines signierten 16-Bit-Displacements modifiziert werden.



Im folgenden Beispiel enthalte A2 den Wert 50000.

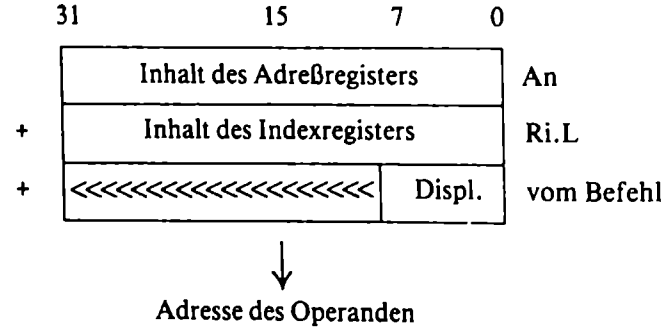
```
CLR.B    100(A2)     Löscht das Byte an Adresse 50100
CLR.B    -32000(A2)  Löscht das Byte an Adresse 18000
```

Diese Adressierungsart eignet sich, wenn das Adreßregister auf eine Datenstruktur zeigt, deren zu adressierende Felder feste Offsets besitzen.

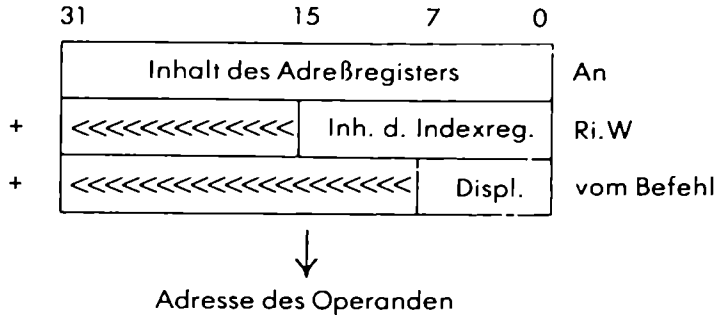
Indirekte Adressierung mit Displacement und Indizierung

Diese Methode ist der vorigen ähnlich, jedoch wird hier noch der Inhalt eines weiteren Registers hinzuaddiert; dabei kann das Displacement nur 8 Bit lang sein.

Langwortformat



Wortformat



Als Indexregister kann ein beliebiges der 16 Register verwendet werden; wieviel vom ihm signifikant ist, wird wie üblich durch das an den Registernamen angehängte Kürzel ('.L', '.W') angegeben.

Im nun folgenden Beispiel enthält A0 $230000, A1 $FFFC (= −4 als 16-Bit-Zahl), und A2 enthält $20.

```
CLR.B    $10(A0,A2)      löscht das Byte $230030
CLR.B    $10(A0,A2.W)      "     "   "   $230030
CLR.B    $10(A0,A2.L)      "     "   "   $230030
CLR.B    $10(A2,A0.L)      "     "   "   $230030
CLR.B    $10(A2,A0.W)      "     "   "       $30
CLR.B    $10(A0,A1.W)      "     "   "   $23000C
CLR.B    $10(A0,A1.L)      "     "   "   $24000C
```

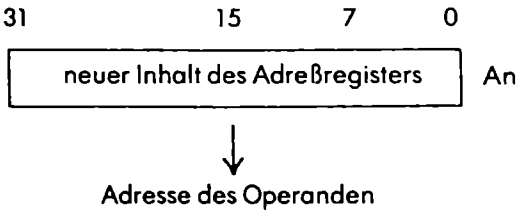
Die Verwendung dieser Adressierungsart erlaubt den indizierten Zugriff auf die Felder einer Datenstruktur, deren Adresse in einem Register vorliegt.

Indirekte Adressierung mit Predecrement oder Postincrement

Es gibt zwei Varianten der indirekten Adressierung, die vor allem den Umgang mit **Stacks** (siehe Kapitel 4) erleichtern. Beide Arten adressieren eine Speicherstelle und ändern den Inhalt des Registers, so daß er auf eine andere Stelle zeigt.

Der **Predecrementmodus** wird in der Form −(An) geschrieben. Bei der Ausführung wird zunächst der Inhalt des Registers je nach der im Befehl angegebenen Operandengröße um 1, 2 oder 4 erniedrigt und dann die durch den neuen Registerinhalt adressierte Speicherstelle angesprochen.

Predecrementmodus

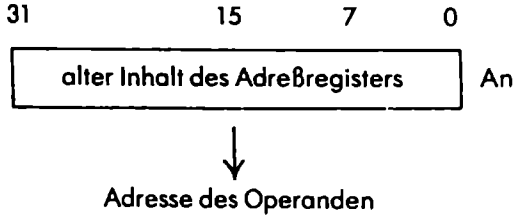


Im folgenden seien A1, A2 und A3 mit 1000 belegt.

```
        CLR.B    -(A1)      setzt A1 auf 999, löscht das
*                           Byte an Adresse 999
        CLR.W    -(A2)      setzt A2 auf 998, löscht das
*                           Wort an Adresse 998
        CLR.L    -(A3)      setzt A3 auf 996, löscht das
*                           Langwort an Adresse 996
```

Das exakte Gegenstück dazu ist der **Postincrementmodus**. Hier wird die durch das Register adressierte Speicherstelle angesprochen; anschließend wird der Inhalt der Registers um 1, 2 oder 4 erhöht.

Postincrementmodus



Enthält A1 1000, so löscht

```
CLR.W    (A1)+
```

das Wort mit der Adresse 1000 und setzt A1 anschließend auf 1002.

Vorsicht ist am Platze, wenn in einer dieser Adressierungsarten das Register A7 verwendet wird. Dieses Register wird von der Hardware in einigen Situationen (Interrupts, Fehler und Unterprogrammaufrufe) verwendet und muß daher stets einen geradzahligen Inhalt haben. Daher wird bei Operationen auf Bytes dieses Register nicht um eins erhöht bzw. erniedrigt, sondern stets wieder auf eine gerade Zahl gesetzt.

Direktoperand

Diese Adressierungsart erlaubt es, den Operanden direkt im Befehl anzugeben und ist daher nur für Quelloperanden erlaubt. Der Wert wird in der Form Zahl bzw. Symbol geschrieben, wobei sich die Länge des Zieloperanden aus dem Kürzel des Befehlscodes ergibt.

```
MOVE.B    #$FF,D0
```

setzt daher das untere Byte von D0 auf $FF.

```
MOVE.L    #$56789ABC,D0
```

hingegen setzt das ganze Register D0 auf 56789ABC (hex).

Ein weitverbreiteter Programmierfehler besteht darin, das Zeichen '#' vor einem Direktoperanden zu vergessen. Hätten Sie versehentlich

```
MOVE.B    $FE,D0
```

eingegeben, so wäre der Inhalt der Speicherstelle $FE anstelle des Wertes $FE ins Register D0 geladen worden.

Einige Befehle haben eine sogenannte »schnelle« Variante im Umgang mit Direktoperanden, die es erlaubt, kleine Werte im Befehlscode unterzubringen. Ein Beispiel dafür ist MOVEQ. Dieser Befehl verwendet einen signierten 8-Bit-Wert:

```
MOVEQ     #-3,D7      Setzt D3 auf -3 (Langwort)
```

Ähnliche Befehle existieren zum Addieren und Subtrahieren von Zahlen zwischen 1 und 8.

```
ADDQ.L    #4,A2
SUBQ.B    #1,(A1)
```

Zusammenfassung der Adressierungsarten

Die nun folgende Tabelle gibt eine kurze Zusammenfassung der oben beschriebenen Adressierungsarten.

Art	*Syntax*	*Operandenadresse*
Datenregister	Dn	Dn
Adreßregister	An	An
Absolute Adressierung	Zahl oder ASYMB	fester Wert (16 oder 32 Bit)
relativ zum PC	RSYMB	[PC] + d16
relativ zum PC mit Index	RSYMB(Ri)	[PC] + [Ri] + d8
Indirekt	(An)	[An]
Indirekt mit Offset	d16(An)	[An] + d16
Indirekt mit Offset und Index	d8(An,Ri)	[An] + [Ri] + d8
Indirekt mit Predecrement	− (An)	[An] := [An] − N ; OA := [An]
Indirekt mit Postincrement	(An)+	OA = [An] ; [An] := [An] + N ;
Direktoperand	# zahl oder # ASYMB	Operand steht direkt im Befehl

Erklärung der Symbole

OA = Operandenadresse	Ri = ein beliebiges Register
Dn = Datenregister n	An = Adreßregister n
d8 = 8-Bit-Displacement	d16 = 16-Bit-Displacement
PC = Programmzähler	N = 1, 2 oder 4
[ ] = »Inhalt von«	:= = »ergibt sich als«
ASYMB = absolutes Symbol	RSYMB = relokatibles Symbol

Implizite Adressierung

Eigentlich handelt es sich hierbei nicht um eine Adressierungsart im eigentlichen Sinne, jedoch ist dies für bestimmte Befehle die einzige Möglichkeit, ihre Operanden zu erhalten. Die implizite Adressierung findet in einigen Befehlen Verwendung, die automatisch bestimmte Register oder Stackplätze verwenden. Solche Register sind der Programmzähler (PC), das Prozessorstatusregister (SR) und die Stackpointer (SP), die durch die beiden Adreßregister 7 (USP und SSP) repräsentiert werden.

Implizite Adressierung wird beispielsweise beim Befehl JMP verwendet, der bei seiner Ausführung den Programmzähler verändert.

2.6 EINTEILUNG DER OPERANDENADRESSEN

Bei den meisten Befehlen sind nicht alle der erwähnten Adressierungsarten erlaubt, da sonst sinnlose oder nicht ausführbare Befehle entstehen würden. Im folgenden finden Sie einige derartige (illegale) Befehle:

```
        JMP     D6          Sprung in ein Register ?
        JMP     -(A5)       Soll A5 um 1, 2 oder 4 erniedrigt
*                           werden ?
        MOVE    D4,#77      Wie kann D4 in eine Konstante
*                           kopiert werden ?
```

Die verschiedenen Adressierungsarten lassen sich in vier (überlappende) Gruppen einteilen. Der Einfachheit halber verwenden wir im folgenden die englischen Bezeichnungen. Die Gruppen sind: **'data references'**, **'memory references'**, **'alterable operands'** und **'control references'**. Wir werden daher im folgenden von einer Adressierungsart vom Typ 'data-alterable' sprechen, wenn diese Adressierungsart zu den Gruppen 'data' und 'alterable' gehört.

Außer dem Inhalt von Adreßregistern kann alles als 'data reference' bezeichnet werden; als 'memory reference' wird alles bezeichnet, was nicht in einem Register vorliegt. 'alterable operands' sind solche, auf die schreibend zugegriffen werden darf. 'control references' sind solche, die zur Adressierung eines Sprungs verwendet werden können.

Die zu jeder Adressierungsart gehörenden Kategorien sind in der folgenden Tabelle zusammengefaßt.

Adressierungsart	*data*	*memory*	*control*	*alterable*
DN	*			*
An				*
relativ zum PC	*	*	*	
rel. + Index	*	*	*	
(An)	*	*	*	*
d16(An)	*	*	*	*
d8(An,Ri)	*	*	*	*
−(An)	*	*		*
(An)+	*	*		*
absolut	*	*	*	*
Direktoperand	*	*		

3
Verschieben und Vergleichen von Daten

Der elementarste Befehl im Befehlssatz des 68000 lautet MOVE. Er verschiebt bei Ausführung Daten von einer Ecke des Computers in eine andere. Anders als bei vielen anderen Computern wird beim 68000 dabei nicht zwischen dem Verschieben von oder nach Registern unterschieden; auch können Daten direkt im Speicher ohne Zuhilfenahme eines Registers verschoben werden.

3.1 EINFACHES VERSCHIEBEN VON DATEN

Der Befehl MOVE besitzt eine Anzahl von Varianten, zu denen wir später kommen werden. Zuerst wollen wir hier ein kleines Programm besprechen, das lediglich einen Speicherbereich mit Daten belegt. Als Quelloperand wird dabei ein Direktoperand verwendet, der Zieloperand wird absolut adressiert.

```
MOVE.B   #123,BYTELOC
```

setzt im Speicher das Byte mit der Adresse BYTELOC auf 123 (dezimal). Als Zieloperand kann auch ein Register verwendet werden; das Programm sieht dann wie folgt aus:

```
MOVE.L   #123,D1
```

Dieser Befehl lädt das Datenregister 1 mit dem Wert 123. Hierbei haben wir anders als im ersten Beispiel die Vollwortform des Befehls verwendet, da das Datenregister 32 Bit breit ist. Hätten wir ein anderes Format wie

```
MOVE.B   #123,D1
```

verwendet, wären zwar die unteren 8 Bits des Datenregisters auf 123 gesetzt worden, die oberen 24 jedoch hätten ihren Wert beibehalten. Dies ist sowohl ein beliebter Programmiertrick als auch ein ebensolcher Fehler. Wenn ein Byte in ein Register geladen wird, sollte dieses vorher tunlichst leer sein. Der gleiche Fehler tritt selbstverständlich auch auf, wenn ein Byte aus dem Speicher in ein Register geladen wird. Der Befehl

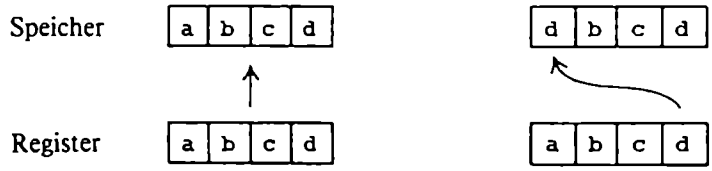
```
MOVE.B   BYTELOC,D1
```

setzt D1 nicht auf das unter der Adresse BYTELOC vorliegende Byte, sondern lädt dieses nur in die unteren 8 Bits. Wenn die folgenden D1 verwendenden Befehle nur Byteformat haben, ist das durchaus in Ordnung, Befehlsfolgen wie

```
MOVE.B  BYTELOC,D1
MOVE.L  D1,LONGLOC
```

setzen jedoch die ersten 3 Byte bei Adresse LONGLOC auf deren ursprüngliche Werte in D1 und in die Adresse LONGLOC+3 das bei BYTELOC vorliegende Byte.

Häufig wird auch vergessen, daß Befehle im Byteformat zwar die unteren (niederwertigsten) Bits eines Registers verändern, jedoch die ersten 8 Bits einer Speicheradresse. Ein Speicherverweis beginnt stets bei der angegebenen Adresse, daher wird beim Verschieben eines Bytes nach LONGLOC ein Byte an dieser Stelle verändert.



Bei Verwendung des Wortformats werden die zwei Bytes bei den Adressen LONGLOC und LONGLOC+1 abgelegt. Die Befehlsfolge

```
MOVE.W  WORDLOC,D1
MOVE.L  D1,LONGLOC
```

verändert die 4 Bytes von LONGLOC bis LONGLOC+3. Die oberen (vorderen) beiden Bytes erhalten den Wert, der ursprünglich in den ersten Bytes von D1 stand, und in LONGLOC+2 und LONGLOC+3 wird das bei WORDLOC liegende Wort abgelegt.

Der Zieloperand eines MOVE-Befehls kann in jeder der Adressierungsarten vom Typ 'data-alterable' angegeben werden, wogegen der Quelloperand mit einer Ausnahme beliebig spezifiziert werden kann: Bei Verwendung des Byteformats darf als Quelloperand kein Adreßregister angegeben werden.

Die Condition-Codes im Statusregister werden bei Ausführung des MOVE-Befehls gesetzt, wenn der Zieloperand eine Speicheradresse oder ein Datenregister ist. Das Z-Bit wird gesetzt, wenn der verschobene Wert gleich Null war, ansonsten wird es gelöscht. Ebenso wird das N-Bit bei negativen Werten gesetzt und sonst gelöscht. V- und C-Bit (Überlauf und Übertrag) werden beide gelöscht. Das X-Bit bleibt unberührt, da es zur Erinnerung an einen Übertrag bei der letzten Rechenoperation dient.

Das Statusregister wird nicht verändert, wenn der Zieloperand ein Adreßregister ist, da die Inhalte der Adreßregister häufig als Indizes verwendet werden. Daher sollten sie ohne Beeinflussung der evtl. in einem späteren Befehl ausgewerteten Statusbits verändert werden können. Um an diesen Unterschied bei der Ausführung eines MOVE-Befehls zu erinnern, wird zum Verschieben von Daten nach Adreßregister ein anderer Befehl verwendet: MOVEA (move address). Der Opcode des Befehls ist derselbe wie bei MOVE, und viele Assembler akzeptieren auch MOVE, wo MOVEA vonnöten wäre. Sie sollten jedoch lieber immer MOVEA verwenden, damit Sie nicht vergessen, daß die Condition-Codes nicht gesetzt werden.

Um eine Anzahl aufeinanderfolgender Speicherplätze mit dem gleichen Wert zu belegen, können wir das folgende kleine Programm schreiben:

```
ORG      $1000
MOVEA.L  #$2000,A0     Lädt Anfangsadresse
MOVE.L   #0,D2         Lädt Konstante 0
MOVE.L   D2,(A0)+      Speichert Wert und schreitet fort
MOVE.L   D2,(A0)+      zur nächsten Stelle
MOVE.L   D2,(A0)+      usw.
END
```

Die erste Programmzeile veranlaßt den Assembler, ab der Adresse $1000 absoluten Code zu generieren. Zeile zwei lädt das Adreßregister mit einem Wert, der im folgenden als Zeiger verwendet wird. Dieser Zeiger hat anfangs den Wert $2000. Analog dazu setzt Zeile drei alle 32 Bits des Datenregisters auf Null. Die Zeilen vier bis sechs verschieben den Inhalt von Datenregister 0 in die durch den Inhalt des Adreßregisters angegebene Speicherstelle. Da der Befehl Vollwortformat hat, werden durch ihn in Zeile vier die 4 Bytes $2000 bis $2003 auf Null gesetzt. A0 wird anschließend um vier erhöht, da der Zieloperand im Postincrementmodus adressiert wird. Bei Verwendung von MOVE.W würde A0 nur um zwei, bei MOVE.B nur um eins erhöht werden. Das Register A0 enthält nun also den Wert $2004, und nach Ausführung von Zeile fünf den Wert $2008. Nach Ablauf des Programms sind die Adressen $2000 bis $200B gelöscht, und in A0 steht der Wert $200C.

3.2 BEDINGTE SPRÜNGE

Wie wir gesehen haben, können bei der Ausführung des MOVE-Befehls die Condition-Codes gesetzt werden. Dies kann verwendet werden, um mit einem kleinen Programm einen großen Speicherbereich zu löschen. Dieses Programm ist vielleicht ästhetisch nicht besonders wertvoll, jedoch läßt sich mit 2 Befehlen nicht sehr viel mehr anstellen. Den zusätzlich zum MOVE benötigten Befehl gibt es bei den meisten Prozessoren: den bedingten Sprung, auch Branch genannt.

Ein bedingter Sprung ist ein Befehl, der in Abhängigkeit von bestimmten Bits im Statusregister die Fortsetzung der Programmausführung an einer anderen Stelle

bewirkt. Entsprechend der verschiedenen Condition-Codes existieren verschiedene **Mnemonics** für die Branch-Befehle. Zunächst einmal sind nur die Befehle BEQ (branch if equal) und BNE (branch if not equal) von Interesse. Der erste bewirkt einen Sprung, wenn das Z-Bit gesetzt ist, der zweite, wenn es gelöscht ist. Wird der Sprung nicht ausgeführt, so wird die Programmausführung mit dem nächsten Befehl hinter dem Sprung fortgesetzt.

Mit diesen beschränkten Mitteln läßt sich nun ein Programm entwerfen, das den Speicher ab einer gegebenen Position abwärts bis zur Adresse Null löscht:

```
        ORG     $1000
        MOVEA.L #$100,A0        Anfangszeiger setzen
LOOP    MOVE.L  #0,-(A0)        Zeiger dekrementieren,
*                               Speicherstelle löschen
        MOVE.L  A0,D1           Zeiger nach D1 kopieren
        BNE     LOOP            Springt zurück, falls noch nicht
*                               fertig
        END
```

Wie im vorherigen Beispiel wird in der ersten Zeile der Programmanfang und in der zweiten der Anfangszeiger festgelegt. In Zeile drei wird in die durch A0 adressierte Speicherstelle der Wert Null geladen. Vor Ausführung des Befehls wird A0 um vier erniedrigt, da der MOVE-Befehl Vollwortformat hat und der Zieloperand im Predecrementmodus adressiert wird. Daher werden bei der ersten Ausführung des Befehls die Adressen $FC bis $FF gelöscht, da A0 nach dem Dekrementieren den Wert $FC enthält. Adresse $100 wird in diesem Programm nicht beeinflußt.

Zeile fünf mutet zunächst etwas seltsam an, da in ihr der Wert von A0 in das im weiteren nicht mehr verwendete Register D1 kopiert wird. Durch diesen Befehl werden jedoch die Condition-Codes beeinflußt, so daß das Z-Bit hierbei gesetzt wird, sobald A0 den Wert Null enthält. Beim ersten Schleifendurchlauf enthält A0 den Wert $FC, das Z-Bit wird also gelöscht. In Zeile sechs wird daher zur weiteren Abarbeitung des Programms ein Sprung nach Zeile drei durchgeführt. A0 ist auch in diesem Durchlauf noch ungleich Null, daher wird dieser Sprung erneut ausgeführt. Durch sukzessives Dekrementieren von A0 in Zeile drei enthält A0 nach dem vierundsechzigsten Durchlauf den Wert Null. In Zeile fünf wird dann das Z-Bit gesetzt, und das Programm endet, da der Sprung in Zeile sechs nicht genommen wird. Zu diesem Zeitpunkt sind dann die Speicherstellen $00 bis $FF gelöscht.

Dieses kleine Programm kann nun auf verschiedene Weisen abgeändert werden. Lautete Zeile drei beispielsweise

```
LOOP    MOVE.B  #0,-(A0)
```

so würde in jedem Schleifendurchlauf nur ein Byte gelöscht. Ansonsten liefe das Programm genauso wie das ursprüngliche, benötigte jedoch viermal soviel Durchläufe und

damit auch mehr Rechenzeit. Diese zusätzliche Zeit wird nur zum Teil dadurch aufgewogen, daß die Ausführung eines MOVE-Befehls im Byteformat schneller geschieht als bei Wort- oder Vollwortformat.

Das neue Programm wäre jedoch kürzer als das erste, da der Direktoperand (#0) nicht ein Vollwort, sondern nur ein Wort im Speicher belegt. Bei diesem Beispiel sind die zwei eingesparten Bytes zu teuer erkauft (oder anders gesehen, die zwei zusätzlichen Bytes ein geringer Preis für die erzielte Geschwindigkeitssteigerung), jedoch muß in der Praxis meist eine schwierigere Entscheidung zwischen Platz- und Zeitoptimierung getroffen werden.

Ohne Geschwindigkeitsverminderung kann das Programm verkürzt werden, indem der Sprungbefehl durch

```
BNE.S    LOOP
```

ersetzt wird. Alle Befehle für bedingte Sprünge besitzen ein langes und ein kurzes Format. In beiden Fällen wird nicht die absolute Adresse des Sprungziels im Programmcode angegeben, sondern eine signierte Zahl gibt an, wie weit (in Byte) vor oder hinter der momentanen Position sich die anzuspringende Adressebefindet. Die lange (standardmäßig gewählte) Befehlsversion verwendet hierfür zwei Bytes, die kurze nur eines. Ohne Angabe des Kürzels '.S' kann ein Sprung daher bis zu 32767 Byte vorwärts oder zurück gehen. Bei Rückwärtssprüngen verwenden einige Assembler automatisch die kurze Form, wenn das Sprungziel nahe genug liegt; bei Vorwärtssprüngen kennt aber kein Assembler die Sprungdistanz und muß daher die lange Version verwenden, wenn der Programmierer nicht explizit das Kurzformat durch Angabe von '.S' gefordert hat.

3.3 VERGLEICHEN VON DATEN

Im vorigen Abschnitt haben wir ausgenutzt, daß die Condition-Codes bei der Ausführung des MOVE-Befehls gesetzt werden. Normalerweise ist das nur dann sinnvoll, wenn der Wert sowieso verschoben werden muß und das Setzen von N- und Z-Bit nur »nützliches Beiwerk« ist. In vielen Fällen müssen wir jedoch nicht nur wissen, ob ein Wert negativ oder Null ist, sondern wir wollen zwei Werte explizit miteinander vergleichen. Genau dies wird durch den Befehl CMP (compare) erreicht.

CMP wird verwendet, um zwei Werte auf Gleichheit zu überprüfen. Sind die beiden Operanden des CMP-Befehls identisch, so wird das Z-Bit gesetzt. Das Programmstück

```
CMP.L    D0,D1
BEQ      EQUAL
```

verursacht somit einen Sprung zum Label EQUAL, wenn D0 und D1 denselben Wert enthalten.

Bei der Ausführung subtrahiert CMP den ersten Operanden vom zweiten und setzt die Condition-Codes (außer X) gemäß dem Resultat. Die beiden Operanden werden dabei nicht verändert, und das Ergebnis der Subtraktion ist nach der Ausführung des Befehls nicht mehr vorhanden.

Der Befehl CMP besitzt vier verschiedene Versionen, wobei die meisten Assembler automatisch die jeweils richtige Form verwenden. CMP darf nur benutzt werden, wenn als Zieloperand ein Datenregister Verwendung findet. Mit einer Ausnahme kann dabei für den Quelloperanden jede beliebige Adressierungsart verwendet werden: Beim Byteformat darf kein Adreßregister benutzt werden.

Bei der Version CMPA wird als Zieloperand ein Adreßregister verwendet, wobei der Befehl nur Wort- oder Vollwortformat besitzen darf; der Quelloperand darf hierbei beliebig adressiert werden. Bei Wortformat wird der Quelloperand sign-extended, bevor die Subtraktion durchgeführt wird. Daher setzt

```
CMPA.W  #$FFFF,A2
```

das Z-Bit, wenn A2 = −1 ($FFFFFFFF) ist, jedoch nicht, wenn in A2 der Wert $FFFF steht.

CMPI kann nur mit einem Zieloperanden vom Typ 'data-alterable' verwendet werden, daher kommen Adreßregister oder PC-relative Adressen nicht in Frage. Der Quelloperand ist hierbei stets ein Direktoperand, und alle drei Längenformate sind gestattet.

```
CMPI.B  #$0A,-(A0)
```

dekrementiert den in A0 enthaltenen Wert um eins und vergleicht dann $0A mit dem durch A0 adressierten Byte. Wird CMPI mit einem Datenregister als Zieloperand verwendet, so ist die Wirkung exakt die von CMP mit einem Direktoperanden als Quelloperand.

Die letzte Version namens CMPM dient zum Vergleichen von Speicherinhalten. Dabei können Quell- und Zieloperand nur im Postincrementmodus adressiert werden, wobei jedoch alle Längenformate erlaubt sind. Große Speicherbereiche können so leicht verglichen werden, wie das folgende Beispiel zeigt. Hierbei werden die 100 Bytes im Speicher ab Adresse $1000 mit denen ab Adresse $2000 verglichen.

```
        MOVEA.L  #$1000,A0       Erster Zeiger
        MOVEA.L  #$2000,A1       Zweiter Zeiger
LOOP    CMPM.B   (A0)+,(A1)+     Vergleich
        BNE.S    NOTSAME         Sprung, falls ungleich
        CMPA.L   #$1000+100,A0   Ende erreicht ?
        BNE.S    LOOP            Nein → erneuter Durchlauf
```

Hier werden zwei Adreßregister als Zeiger auf die zu vergleichenden Datenbereiche verwendet. In Zeile drei werden der Vergleich durchgeführt und die Zeiger inkrementiert. Falls sich dabei Ungleichheit herausstellt, springt das Programm in Zeile vier aus der Schleife zum Label NOTSAME; ansonsten wird in Zeile fünf geprüft, ob bereits alle Speicherstellen verglichen wurden, und in Zeile sechs ggf. zurück zum Schleifenanfang gesprungen. Wird der Sprung nach NOTSAME nicht durchgeführt, sondern das Programm hinter Zeile sechs fortgeführt, so waren die beiden Speicherbereiche identisch.

3.4 NOCH MEHR BEDINGTE SPRÜNGE

Die bisher behandelten bedingten Sprünge prüfen nur zwei Bedingungen ab. BEQ bewirkt einen Sprung, wenn das Z-Bit gesetzt ist, BNE, wenn es nicht gesetzt ist. Sie haben jedoch sicher bereits erraten, daß es noch eine Anzahl weiterer Versionen des Bcc-Befehls gibt, die andere Bedingungen abprüfen.

Die erste Gruppe dieser Sprungbefehle behandelt nur ein Bit im Statusregister. So wie BEQ und BNE einen Sprung in Abhängigkeit vom Zustand des Z-Bits verursachen, springen BCS (branch on carry set) und BCC (branch on carry clear) in Abhängigkeit vom C-Bit. BCS springt, wenn es gesetzt, und BCC, wenn es gelöscht ist.

Zur Behandlung des N-Bits dienen BMI (branch on minus, springt bei gesetztem N-Bit) und BPL (branch on plus, springt bei gelöschtem N-Bit). War das Ergebnis einer Operation Null, so wird das N-Bit gelöscht. Null gehört also beim 68000 (wie bei den meisten Prozessoren) zu den positiven Zahlen.

Als letztes Paar in dieser Gruppe stehen BVS (branch on overflow set) und BVC (branch on overflow clear), die das V-Bit prüfen.

Die Befehle der zweiten Gruppe der bedingten Sprünge prüfen mehrere Bedingungen, bevor ein Sprung unternommen wird. Einige dieser Befehle ähneln denen aus der ersten Gruppe der einfachen Befehle, von denen sie sich nur durch die Behandlung des Überlauf- und Übertrag-Flags unterscheiden. Viele Befehle (wie z.B. MOVE) löschen sowieso das V- und das C-Bit, so daß hierbei die Befehle dieser beiden Gruppen identische Verhaltensweisen zeigen. Die Unterschiede offenbaren sich erst bei der Behandlung von Zahlen mit Vorzeichen.

Beim Vergleich von signierten Zahlen finden die Befehle BLT (branch if less than) und BGE (branch if greater than or equal) Verwendung. BLT springt nur dann, wenn das N-Bit gesetzt und das V-Bit gelöscht, oder aber das N-Bit gelöscht und das V-Bit gesetzt ist. Solange also kein Überlauf auftrat, wirkt BLT wie BMI; trat ein Überlauf auf, so wirkt er wie BPL. BGE springt, wenn N- und V-Bit beide gesetzt oder beide gelöscht sind, wirkt also wie BPL ohne bzw. wie BMI bei Überlauf.

Die Befehle BLS (branch if low or same) und BHI (branch if high) prüfen Z- und C-Bit. BLS springt, wenn Z- oder C-Bit gesetzt sind, und BHI, wenn beide gelöscht sind. Die Befehle BCC und BCS werden daher mitunter auch als BHS (branch if high or same) und BLO (branch if low) bezeichnet.

Die kompliziertesten der bedingten Sprungbefehle sind BLE (branch if less or equal) und BGT (branch if greater than). BLE springt, wenn auch BLT springen würde, aber auch, wenn das Z-Bit gesetzt ist. BGT springt nur, wenn BGE springt und zusätzlich das Z-Bit gelöscht ist.

Wie wir später sehen werden, finden die gleichen Bezeichnungen für Bedingungen bei den Befehlen DBcc und Scc Verwendung. Bei diesen Befehlen gibt es zusätzlich die Bedingungen T (true) und F (false). Der Befehl, der eigentlich BT heißen müßte, heißt jedoch BRA (branch), wogegen es BF nicht gibt, da »Springe nie« ein völlig sinnloser Befehl wäre.

3.5 EIN EINFACHES PROGRAMM ZUR SPEICHERDIAGNOSE

Inzwischen kennen wir bereits genügend Befehle, um ein kleines Programm zur Speicherdiagnose schreiben zu können. Dazu schreiben wir in einen Speicherbereich ein bestimmtes Bitmuster und kontrollieren anschließend, ob dieses Muster auch wieder lesbar ist. Damit läßt sich feststellen, ob alle Speicherbausteine des Rechners funktionieren. Da wir noch nichts über Ein- und Ausgabe gelernt haben, springt das Programm bei Entdeckung eines Fehlers zu einem Label. An dieser Stelle könnte dann eine Fehlermeldung ausgegeben werden; dies interessiert uns jedoch im Augenblick nicht weiter.

```
          ORG       $400
*
* Konstantendefinitionen
*
MEMLO     EQU       $1000          untere Grenze
MEMHI     EQU       $2000          obere Grenze
TPAT      EQU       $AA            Testmuster
ERRMSG    EQU       $3000          Fehlermeldung

* Das eigentliche Programm

ENTER     MOVEA.L   #MEMLO,A0      setzt Anfangszeiger

* Speicher mit Testmuster beschreiben

LP1       MOVE.B    #TPAT,(A0)+    Wert speichern, A0 erhöhen
          CMPA.L    #MEMHI,A0      Obere Grenze erreicht ?
          BNE.S     LP1            Nein → weitermachen

* Ist das Muster noch vorhanden ?
```

```
        MOVEA.L   #MEMLO,A0       Anfangszeiger wiederherstellen
LP2     CMPI.B    #TPAT,(A0)+     Ist das Muster noch da ?
        BNE       ERRMSG          Nein → Fehlermeldung ausgeben
        CMPA.L    #MEMHI,A0       Obere Grenze erreicht ?
        BNE.S     LP2             Nein → weitermachen

* Test beendet, zurück zum Anfang

        BRA.S     ENTER
        END
```

In den ersten paar Zeilen werden hier der Programmanfang sowie einige Konstanten festgelegt. Dieser Definition mittels EQU ist der Verwendung von Zahlen in Direktoperanden stets der Vorzug zu geben. Zum einen liest sich das Programm durch die mnemonische Wirkung der Namen leichter, und zum anderen sind Änderungen des Programmes leichter durchführbar. Unser Programm prüft den Speicherbereich $1000 bis $1FFF, oder anders ausgedrückt, den Bereich von MEMLO bis MEMHI−1. Zum Prüfen eines anderen Speicherbereiches müssen daher nur diese Konstanten neu definiert werden; es sind also keine Änderungen im Programm selbst nötig.

Der Einsprung in das Programm erfolgt beim Label ENTER, wo A0 auf die Anfangsadresse des zu prüfenden Bereiches gesetzt wird. Das Label LP1 stellt den Schleifenanfang dar, das Testmuster wird hier an die durch A0 adressierte Stelle geschrieben, und A0 wird erhöht. In der nächsten Zeile wird mit CMPA getestet, ob bereits der gesamte zu prüfende Speicher mit dem Testmuster belegt wurde. Ist das nicht der Fall, springen wir nach LP1 zurück; andernfalls geht der Programmablauf zur Testschleife über.

Dort wird zunächst der Anfangszeiger wiederhergestellt und dann in einer ähnlichen Schleife wie oben die durch A0 adressierte Speicherstelle mit dem Testmuster verglichen. Treten dabei Unterschiede auf, so springen wir zum Label ERRMSG, wo eine Fehlermeldung ausgegeben wird. Dort wäre ggf. zu beachten, daß A0 nach dem Vergleich bereits inkrementiert wurde und somit die Adresse der defekten Speicherstelle gleich dem um eins verminderten Inhalt von A0 ist.

In diesem Beispiel haben wir die Vollwortformate von MOVEA und CMPA verwendet, obwohl die Wortformate hinreichend gewesen wären. Sie hätten das Programm sogar kürzer und schneller gemacht. Bei einer späteren Programmänderung hätte dies jedoch leicht einem Mitmenschen zum Verhängnis werden können. Sollte jemand die obere Grenze von $2000 nach $8000 verschieben wollen, so würde er nach einem schnellen Blick auf das Programm nur den Wert des Symbols MEMHI von $2000 auf $8000 ändern. Bei Verwendung des Wortformats von MOVEA und CMPA würde beim Test auf das Ende des Bereichs MEMHI als Direktoperand mit A0 verglichen. Da durch die Verwendung des Wortformats der Direktoperand zunächst auf 32 Bit signextended wird, gelingt dieser Test nur dann, wenn A0 den Wert $FFFF8000 enthält, wobei dies keine gültige Adresse darstellt. Das Programm würde jedoch bereits vorher

terminieren, da ein Busfehler auftritt, wenn der gültige Adreßbereich erschöpft ist. Man sollte daher stets die Vollwortformate der Befehle verwenden, wenn es darum geht, Adressen in Adreßregister zu laden. Die anderen Formate eignen sich eigentlich nur dann, wenn Adreßregister zur Speicherung von Datenwerten verwendet werden sollen.

3.6 DIE NULL

Zum Umgang mit dem Wert Null gibt es im Befehlssatz des 68000 zwei Spezialbefehle. Zwar haben wir bereits gesehen, daß MOVE und CMP als Direktoperanden den Wert Null verwenden können, jedoch benötigt ein Direktoperand 16 bzw. 32 Bit Platz im Code hinter den 16 Bit für den Befehlscode. Der Befehl

```
MOVE.L  #0,D0
```

benötigt also 16 Bit für den Befehlscode und 32 Bit für die Darstellung des Wertes Null. Da häufig Werte auf Null gesetzt werden müssen, gibt es hierfür zwei Befehle, die jeweils nur 16 Bit lang sind.

Als erstes ist hier CLR zu nennen, der bei Ausführung die angegebene Stelle auf Null setzt. Diese Stelle muß 'data-alterable' adressiert sein, und somit läßt sich dieser Befehl nicht zum Löschen eines Adreßregisters verwenden. Ansonsten jedoch kann mit ihm ein Byte, Wort oder Vollwort im Speicher bzw. die unteren 8, 16 oder alle 32 Bits eines Datenregisters auf Null setzen.

Die Condition-Codes werden so gesetzt, als wäre zum Löschen der MOVE-Befehl mit Direktoperand Null verwendet worden, d.h. X bleibt unverändert, Z wird gesetzt und die anderen werden gelöscht.

Vorsicht ist bei diesem Befehl angebracht, wenn die angesprochene Speicherstelle Teil der **I/O-Page** ist. Dort liegen die Speicherstellen der peripheren Geräte (memory mapping), sie erscheinen dem Prozessor aber als Teil seines eigenen Speichers. Der Prozessor liest zunächst den Wert der angegebenen Stelle, bevor er ihn durch Null ersetzt. Wenn allein schon das Lesen einer Adresse Auswirkungen auf ein Peripheriegerät hat, kann dies zu den seltsamsten Effekten führen.

Der Befehl TST dient dazu, einen Wert mit Null zu vergleichen. Auch hier muß der Operand 'data-alterable' adressiert sein, und alle Längenformate sind zulässig. Ist der Wert des Operanden gleich Null, so wird das Z-Bit gesetzt, ansonsten wird es gelöscht. Anschließend kann wie nach CMP mit einem bedingten Sprung das Ergebnis des Tests ausgenutzt werden.

Mittels TST kann auch geprüft werden, ob ein Wert negativ ist. In diesem Fall wird das N-Bit gesetzt, ansonsten wird es gelöscht. Das X-Bit ändert sich nicht, V- und

C-Bit werden gelöscht. BMI und BPL können also auch nach einem Aufruf von TST verwendet werden, um das Resultat auszuwerten. Da das C-Bit nach TST immer gelöscht ist, haben BLT und BMI den gleichen Effekt, ebenso BGT und BPL.

3.7 KLEINE ZAHLEN

Viele Programmierer werden das Vollwortformat so oft wie möglich verwenden wollen, da die Möglichkeit des Umgangs mit 32-Bit-Werten den 68000 so sehr von seinen Rivalen unterscheidet. Häufig ist es jedoch auch nötig, ein Register mit Null oder einer kleinen Zahl zu belegen. Für den ersten Fall kennen wir bereits den Befehl CLR, der ein Register oder eine Speicherstelle löscht.

Um ein Register auf einen kleinen Wert zu setzen, kann dieser als Direktoperand eines MOVE-Befehls im Vollwortformat spezifiziert werden. Nachteilig dabei ist jedoch, daß dieser Befehl 6 Bytes benötigt: 2 für den Befehlscode und 4 für den Direktoperanden. Natürlich werden 4 Byte benötigt, um einen Direktoperanden für einen Befehl im Vollwortformat darzustellen, dennoch stört diese haarsträubende Verschwendung von Platz, wenn der Operand auch in ein einziges Byte passen würde.

Um auch diese Situation zu meistern, enthält der Befehlssatz eine Spezialform des MOVE-Befehls: MOVEQ (move quick). Dieser kann nur Vollwortformat haben und nur zum Verschieben einer Zahl verwendet werden, die in ein Byte eines Datenregisters paßt. Die Wirkung der Ausführung von MOVEQ entspricht exakt der Ausführung von MOVE mit einem Direktoperanden zwischen −128 und +127 und einem Datenregister als Zieloperanden, jedoch benötigt MOVEQ nur zwei Bytes. Der Direktoperand wird dabei in das untere Byte des Befehlscodes codiert. Das Datenregister wird vollständig verändert, wobei die vorderen 24 Bit durch Sign-Extension gesetzt bzw. gelöscht werden. N- und V-Bit werden je nach dem Operanden gesetzt oder gelöscht, V- und C- Bit generell gelöscht und das X-Bit bleibt unberührt.

Der Befehl MOVEQ benötigt bei der Ausführung weniger Zeit als CLR mit Vollwortformat, daher eignet er sich besser dazu, ein Datenregister zu löschen. Vergessen Sie aber nicht, daß MOVEQ stets Vollwortformat hat und nur zum Laden von Datenregistern verwendet werden kann.

3.8 TESTEN EINZELNER BITS

Viele Befehle operieren mit einzelnen Bits. Sie werden alle in Kapitel 6 im Detail erklärt; einer von ihnen ist jedoch bereits hier von Interesse. Der Befehl BTST (bit test) prüft ein einzelnes Bit des angegebene Operanden. Ist dieses Bit gesetzt, so wird auch das Z-Bit der Statusregisters gesetzt, ansonsten wird dieses gelöscht. Die anderen Statusbits bleiben unbeeinflußt.

Der in jeder Adressierungsart vom Typ 'data' angebbare Operand wird nicht verändert. Die bei Ausführung des Befehls durchgeführten Operationen hängen davon ab, ob es sich beim Operanden um eine Speicherstelle oder um ein Datenregister handelt. Im ersten Fall wird ein Byte aus dem Speicher gelesen und ein Bit von ihm getestet. Das niederwertigste Bit erhält dabei die Ordnungszahl 0, das höchstwertige 7. Zahlen größer als 7 werden modulo 8 verwendet.

Wird ein Datenregister als Zieloperand angegeben, so können die Bitangaben zwischen 0 und 31 liegen. Auch hier werden größere Zahlen modulo 32 gerechnet. In Abhängigkeit vom Zieloperanden wird der BTST-Befehl im Byte- oder Vollwortformat verwendet, wobei dies vom Programmierer nicht beeinflußbar ist.

Die Ordnungszahl des gewünschten Bits wird als Quelloperand spezifiziert, wobei nur zwei Adressierungsarten möglich sind. Im einfacheren Fall wird ein Direktoperand verwendet, der (evtl. modulo gerechnet) die Nummer des Bits angibt, oder aber es wird ein Datenregister angegeben. In diesem Fall nimmt der Prozessor den in diesem Register vorliegenden Wert (evtl. modulo) als Bitnummer.

Der Quelloperand wird (anders als bei den Befehlen anderer Prozessoren) immer als Nummer eines Bits interpretiert und nicht als Bitmuster, welches das zu testende Bit maskiert.

3.9 TESTEN VON BEDINGUNGEN

Wir haben bereits die Bcc-Familie behandelt, deren Mitglieder Sprünge in Abhängigkeit von verschiedenen Bedingungen verursachen. Zwar ist dies die häufigste Verwendung der Condition-Codes, aber es gibt noch zwei weitere Befehle, welche die Condition-Codes benutzen. Als erster ist hierbei der Befehl Scc (set according to condition code) zu erwähnen. Dieser prüft zunächst die Bits des Statusregisters auf die gleiche Weise wie der bereits bekannte Bcc-Befehl und setzt anschließend seinen Operanden auf $FF, wenn die Bedingungen erfüllt waren, und auf Null, falls dies nicht der Fall war. Die Zeile

```
          Scc       BYTELOC     BYTELOC entsprechend der
*                                Bedingung setzen
```

ersetzt daher das folgende Programmstück:

```
          Bcc       NXT         Sprung, falls Bedingung erfüllt
          CLR.B     BYTELOC     Falls nicht → BYTELOC löschen
          BR.S      NXT1        Sprung zum Ende
NXT       MOVE.B    #$FF,BYTELOC   BYTELOC auf $FF setzen
NXT1      ....                  Rest des Programms
```

Scc kann nur zum Setzen einzelner Bytes verwendet werden, und diese müssen 'data-alterable' adressiert sein. Scc eignet sich auch dafür, ein Byte immer auf $FF zu setzen, indem die Bedingung TRUE (wahr) abgeprüft wird:

```
ST        BYTELOC     BYTELOC immer auf $FF setzen
```

Die Verwendung der Bedingung FALSE (falsch) entspricht der Verwendung des Byteformats des CLR-Befehls.

```
SF        BYTELOC
```

ist äquivalent zu:

```
CLR.B     BYTELOC
```

Meistens wird der Scc-Befehl verwendet, um den Zustand bestimmter Bits des Statusregisters für spätere Verwendung zu speichern.

3.10 SCHLEIFEN

Sehr oft wird in einem Computer eine Anzahl von Befehlen immer und immer wieder ausgeführt. Diese sogenannte Schleife wird normalerweise durch eine Iterationsvariable kontrolliert, die solange erhöht wird, bis sie einen bestimmten Endwert erreicht oder überschritten hat.

Der 68000 besitzt einen Befehl zur Vereinfachung der Schleifenbildung. Dieser arbeitet der oben geschilderten Methode genau entgegengesetzt, d.h. er **dekrementiert** eine Iterationsvariable und beendet die Schleife, sowie diese Variable negativ wird.

Diese Befehlsfamilie wird mit DBcc (decrement and branch on condition codes) bezeichnet. Bei der Ausführung wird zunächst die Bedingung überprüft, und nur bei erfolglosem Test der Dekrement- und Sprungteil des Befehls ausgeführt. Zunächst wollen wir den Fall betrachten, daß die Bedingung niemals erfüllt ist. Der entsprechende Befehl heißt (wie erwartet) DBF. Dieser Befehl ist der am häufigsten verwendete seiner Familie und wird von den meisten Assemblern auch als DBRA (decrement and branch) akzeptiert.

Als Quelloperanden benötigt DBRA ein Datenregister, als Zieloperanden ein Label. Der Befehl verwendet nur das Wortformat. Ist der Inhalt des Datenregisters Null, so wird mit der Programmausführung beim nächsten Befehl hinter DBRA fortgefahren, ansonsten wird der Registerinhalt um eins erniedrigt und ein Sprung zum angegebenen Label veranlaßt.

Das Dekrementieren des Registers geschieht immer, auch wenn der Sprung nicht ausgeführt wird, so daß das Register zum Abschluß der Schleife nicht den Wert Null enthält. Da der Befehl nur das Wortformat verwendet, werden nur die unteren 16 Bits für den Zähler verwendet. Das angesprungene Label kann im Programm vor oder hinter dem DBRA-Befehl stehen, jedoch ist die zweite Möglichkeit relativ sinnlos.

```
        MOVE.L   #$2000,A1  Zeiger setzen
        MOVE.W   #19,D0     Zähler setzen
LOOP    CLR.B    (A1)+      Byte löschen, Zeiger weitersetzen
        DBRA     D0,LOOP    Solange D0 >= 0 gilt
```

In diesem Beispiel finden die unteren 16 Bits von D0 als Zähler Verwendung. Dieser wird anfangs auf 19 gesetzt, und A1 erhält einen Zeiger auf einen Speicherbereich. Beim Label LOOP wird das durch A1 adressierte Byte gelöscht und der Zeiger um eins weitergesetzt. Der DBRA-Befehl dekrementiert D0 und überprüft, ob das Ergebnis negativ war. Ist das nicht der Fall, wird zum Label LOOP zurückgesprungen und die Ausführung dort wieder aufgenommen. Dies geschieht immer wieder, bis D0 durch das Dekrementieren einen negativen Wert erhält. Der Sprung wird nun nicht ausgeführt und das Programmstück wird verlassen. Die unteren 16 Bits von D0 enthalten nun den Wert $FFFF, A1 enthält den Wert $2014.

Oft ergibt sich die gewünschte Anzahl an Schleifendurchläufen als das Ergebnis einer früheren Berechnung, wobei die Anzahl Null (also kein Schleifendurchlauf) durchaus auch auftreten kann. Hierbei empfiehlt es sich, das Zählerregister vor dem Schleifenbeginn zu initialisieren und einen unbedingten Sprung zum DBRA-Befehl am Schleifenende durchzuführen. Dies hat zusätzlich den Vorteil, daß im Zählregister die tatsächliche Durchlaufzahl anstelle der um eins erniedrigten abgelegt werden kann.

Der DBRA-Befehl ist zwar der meistverwendete seiner Gattung, jedoch ist die volle Form des DBcc-Befehls noch wesentlich leistungsfähiger. Hierbei wird die Schleife nicht nur beendet, wenn der Zähler negativ wird, sondern auch, wenn die spezifizierte Bedingung erfüllt ist. In diesem Fall wird auch der Zähler nicht noch einmal dekrementiert.

Durch Verwendung der DBcc-Befehle ergeben sich eine Anzahl sehr leistungsfähiger Schleifenkonstrukte. Sei beispielsweise das Problem gegeben, Daten solange aus einem Bereich in einen anderen zu kopieren, bis im Ursprungsbereich ein bestimmtes Byte vorgefunden wird. Der Zielbereich habe nur eine begrenzte Länge, so daß das Kopieren auch bei Erreichen dieser Grenze beendet werden muß. Derartige Probleme treten auf, wenn etwa eine Zeile vom Terminal in einen internen Puffer kopiert werden muß. Das Kopieren muß beendet werden, wenn das Return-Zeichen (Zeilenende) vorgefunden wird oder wenn eine Zeile länger ist als erlaubt. Zu diesem Zweck können wir das folgenden Programmstück verwenden:

```
CR      EQU     $0D         Zeilenende-Zeichen (Return)
        MOVEA.L #$2000,A3   Zeiger auf internen Puffer
        MOVE.W  #79,D0      max. 80 Zeichen
RCH     ...                 Irgendwie ein Zeichen von der
*                           Tastatur nach D1 holen
        MOVE.B  D1,(A3)+    Im Puffer abspeichern
        CMP.B   #CR,D1      Zeichen = Return ?
        DBEQ    D0,RCH      Bis Return oder Puffer voll
```

Die beiden ersten Zeilen initialisieren A3 als Zeiger auf den Puffer und D0 als Zählvariable. In jedem Schleifendurchlauf wird ein Zeichen von der Tastatur irgendwie nach D1 gelesen und anschließend im Puffer abgelegt. Der Pufferzeiger wird dabei um eins erhöht. Abschließend wird das eingelesene Zeichen mit dem Zeilenende-Zeichen verglichen. Der Befehl DBEQ führt den 'decrement and branch'-Teil nur dann aus, wenn dieser Vergleich nicht gelang; ansonsten wird die Schleife direkt verlassen. Anderenfalls wird wieder in die Schleife eingesprungen, sofern noch Platz im Puffer vorhanden ist.

3.11 EINFACHE EIN- UND AUSGABE

Bei allen bisherigen Beispielen gingen wir stillschweigend davon aus, daß unsere Testprogramme unter der Kontrolle irgendeines Monitors liefen, der uns die Eingabe und den Start unserer Programme erlaubt. Normalerweise gibt uns ein solcher Monitor auch die Möglichkeit, Informationen an ein am Computer angeschlossenes Terminal auszugegeben und Eingaben von diesem einzulesen.

Unser Programm kann auch unter der Kontrolle eines Betriebssystems ablaufen, wobei dieses dann für die Kommunikation mit der Umwelt verantwortlich ist. In beiden Fällen ist es wahrscheinlich, daß diese Kommunikation über eine serielle Leitung vonstatten geht. An eine solche Leitung kann leicht ein passendes Terminal angeschlossen werden. Eine serielle Leitung wird meist durch den Einsatz eines Spezialbausteins namens **ACIA** (asynchronous communication interface adapter, asynchrone Kommunikationsschnittstelle) realisiert. Die folgende Beschreibung orientiert sich am ACIA 6850, jedoch arbeiten mehr oder wenige alle I/O-Bausteine in einer ähnlichen Manier. Wir wollen im folgenden nicht zu tief ins Detail gehen; hier sei nur gesagt, daß dieser Baustein alles bewerkstelligt, was zum Senden und Empfangen einzelner Bytes über eine serielle Schnittstelle benötigt wird.

Ein ACIA wird über den normalen Adreßbereich des 68000 adressiert (memory mapped) und besitzt normalerweise 2 Ports: einen Daten- und einen Kontrollport. Bei einem 8-Bit-Rechner erscheinen diese beiden Ports an zwei aufeinanderfolgenden Adressen; beim 68000 werden sie als die unteren Bytes zweier Worte adressiert.

Zu Beginn muß der ACIA durch Schreiben des Wertes 3 in den Kontrollport initialisiert werden. Laut Manual muß anschließend eine gewisse Zeit gewartet werden, damit sich der ACIA wieder beruhigen kann.

Als nächstes müssen die Charakteristika der seriellen Leitung ausgewählt werden. Darunter fallen Parität, Interruptverhalten und ähnliches mehr. Wir werden den ACIA zunächst nur im 'polled'- Modus betreiben. In diesem Modus fragt der Prozessor zyklisch beim ACIA an, ob zwischenzeitig ein Byte über die serielle Leitung empfangen wurde. Liegen zwischen diesen Anfragen zu große Zeiträume, kann es auch passieren, daß zwischendurch Informationen verlorengehen. Der ACIA ist zwar intelligent genug, dies zu bemerken und es dem Prozessor mitzuteilen; es gibt aber keine Möglichkeit, diese verlorenen Bytes wiederzugewinnen. Später einmal werden wir einen ACIA im Interrupt-Modus betreiben, aber für diesmal soll der 'polled'-Modus ausreichen. Der Wert $15 dient uns als Setup für den ACIA, wobei wir die Bedeutung dieses Werts nicht weiter erläutern wollen. Sollten Sie jemals einen ACIA initialisieren müssen, werden Sie in dessen Manual die genaue Bedeutung finden. Dieser Wert wird einfach in den Kontrollport geschrieben.

Der Zustand des Datenports kann nach Initialisierung des ACIAs an den beiden unteren Bits des Kontrollports erfragt werden. Bit 0 ist gesetzt, wenn ein Byte über die serielle Leitung empfangen wurde. Durch Lesen dieses Bytes vom Datenport wird dann dieses Bit wieder gelöscht und erst durch das nächste eintreffende Zeichen wieder gesetzt. Bit 1 signalisiert, daß der ACIA in der Lage ist, ein Zeichen auszusenden. Ist dieses Bit gesetzt, so können wir einen Wert in den Datenport schreiben, der dann vom ACIA über die Leitung ausgesendet wird. Dies benötigt etwas Zeit, und der ACIA löscht für diesen Zeitraum Bit 1. Nach Übertragung des Bytes wird das Bit wieder gesetzt, und wir können ein weiteres Zeichen abschicken. Senden und Empfangen geschehen dabei völlig unabhängig voneinander, daher das 'asynchron' im Namen des Bausteins.

Das hört sich vielleicht alles etwas kompliziert an, aber Sie werden noch merken, wie einfach ein ACIA zu handhaben ist. Wir wollen nun einfach mal den Text 'Hallo !' auf das Terminal ausgeben.

```
* Für den AICA benötigte Werte

A_RTS     EQU       $03          Reset-Code
A_INIT    EQU       $15          Setup-Wert
A_RDY     EQU       1            Bereitschafts-Bit im Kontrollport
A_CTRL    EQU       $840021      Adresse des Kontrollports
A_DATA    EQU       $840023      Adresse des Datenports

          ORG       $1000

* Initialisierung des ACIAs

ENTER     MOVE.B    #A_RST,A_CTRL  ACIA rücksetzen
          MOVE.W    #1000,D0       Zähler initialisieren
WAIT      DBRA      D0,WAIT        Warteschleife
          MOVE.B    #A_INIT,A_CTRL Setup des ACIAs

* Senden des Textes über die serielle Leitung
```

```
          MOVEA.L   #STRING,A0      Zeiger auf den Text
NXT       BTST      #A_RDY,A_CTRL   Bereit zur Übertragung ?
          BEQ.S     NXT             Nein -> warten
          MOVE.B    (A0)+,A_DATA    Zeichen auf den Datenport
          TST.B     (A0)            Ist das nächste Byte Null ?
          BNE.S     NXT             Nein -> weiter ausgeben

* Datenbereich für den Text

STRING    DC.B      'Hallo !'       Text
          DC.B      0               Endmarkierung
```

In den ersten Zeilen definieren wir uns einige nützliche Symbole, darunter den Reset- und den Setupwert für den ACIA sowie die Adressen seines Kontroll- und Datenports. Die Ausführung des Programms beginnt beim Label ENTER, wo wir den Resetwert in den Kontrollport schreiben. Da wir anschließend eine Weile warten müssen, verwenden wir D0 als Zähler und zählen in der mit WAIT markierten Zeile so lange friedlich vor uns hin, bis daß der Wert in D0 durch Ausführung des DBRA-Befehls von 1000 bis unter Null gefallen ist. Daraufhin schreiben wir noch den Setupwert in den Kontrollport. Die serielle Leitung sollte nun fertig zur Benutzung sein.

Der Befehl in der Zeile vor dem Label NXT setzt A0 auf den Wert des Symbols STRING und damit (wie Sie weiter unten sehen können) auf die Adresse des auszugebenden Textes. Als Abschlußmarkierung des Textes haben wir hier das Byte Null verwendet.

Der BTST-Befehl in der mit NXT bezeichneten Zeile prüft das Bereitschafts-Bit des Kontrollports. Solange dieses nicht gesetzt ist, wird bei diesem Test das Z-Bit gesetzt und somit in der folgenden Zeile an den Anfang dieser Schleife gesprungen. In dieser Schleife hängen wir so lange fest, bis daß der ACIA Bereitschaft signalisiert. Dann schreiben wir das durch A0 adressierte Zeichen in den Datenport, woraufhin dieses Zeichen ausgesandt wird. A0 wird anschließend um eins erhöht und zeigt nun auf das nächste auszugebende Zeichen bzw. auf die Abschlußmarkierung. Durch Verwendung des Befehls TST prüfen wir nun, ob das durch A0 adressierte Byte den Wert Null hat. Ist dies nicht der Fall, so springen wir in der folgenden Zeile wieder an den Anfang der Ausgabeschleife zurück, um die restlichen Zeichen auszugeben. Andernfalls endet unser Programmstück hier.

Es ist normalerweise nicht sonderlich sinnvoll, nach der Übergabe eines Zeichens an den ACIA darauf zu warten, daß dieser für die nächste Übertragung bereit ist. Da der ACIA intern unabhängig vom 68000 arbeitet, ist es besser, den Prozessor zwischendurch an etwas anderem arbeiten zu lassen und nur ab und zu den Zustand des ACIAs zu prüfen, um dann ggf. das nächste Zeichen zu übergeben.

Nachdem der Prozessor in unserem obigen Programm den Text ausgegeben hat, wird er versuchen, den nächsten Befehl hinter BNE auszuführen. In diesem Fall wird dieser durch den Text spezifiziert, der als Befehl interpretiert vermutlich ziemlich unsin-

nig ist. Normalerweise erfolgt daher zum Abschluß des Programms ein Rücksprung in den Monitor bzw. in das Betriebssytem. Wie dieser auszusehen hat, wird später noch erklärt.

3.12 DATENAUSGABE AN PERIPHERIEGERÄTE

Wie wir sahen, besitzt ein ACIA zwei Ports, die als die unteren Bytes zweier benachbarter Worte adressiert werden. Die Spezifikation eines ACIAs besagt, daß die Ports normalerweise in aufeinanderfolgenden Speicherstellen liegen. Da der ACIA ursprünglich für 8-Bit-Prozessoren entwickelt wurde, ist dies bei Verwendung des ACIAs in einem derartigen Rechner auch korrekt. Beim 68000 jedoch müssen die Ports wegen des 16 Bit breiten Datenbusses in durch eine ungerade Anzahl von Bytes getrennten, sogenannten **alternierenden** Speicherstellen liegen. Dies gilt nicht nur für unseren ACIA, sondern auch für jedes andere Peripheriegerät, das ursprünglich einmal für 8-Bitter entworfen wurde.

Meistens stört diese Einschränkung nicht besonders, da einzelne Bytes im Speicher ohne weiteres gelesen und beschrieben werden können. Mitunter jedoch führt es zu Umständlichkeiten und damit zur Verlangsamung des Programmablaufs. Daher wurde eine Spezialversion des MOVE-Befehls implementiert, die sich mit diesem Problem herumschlägt. Der Befehl MOVEP (move peripheral) erhält als Operanden ein Datenregister sowie eine durch Displacement und Adreßregister spezifizierte Speicherstelle. Ist das Datenregister der Quelloperand, so wird sein Inhalt Byte für Byte in alternierende Speicherstellen geladen, wobei an der spezifizierten Speicherstelle begonnen wird. MOVEP kann nur im Wort- oder Vollwortformat verwendet werden; das Byteformat entspräche schließlich dem einfachen MOVE-Befehl.

```
MOVE.L   #$01020304,D1   Daten
MOVEA.L  #$C00000,A1     Adresse
MOVEP.L  D1,1(A1)        Abspeichern
```

In diesem Beispiel gehen wir davon aus, daß vier Ports für irgendwelche Peripheriegeräte in den Adressen $C00001, $C00003, $C00005 und $C00007 vorliegen. Der MOVEP-Befehl legt bei der Ausführung zunächst das oberste Byte des Quelloperanden (hier $02) in der angegebenen Speicherstelle ($C00001) ab. Das nächste Byte ($02) wird in der übernächsten Speicherstelle abgelegt, hier also bei $C00003. Das dritte Byte wird dann in $C00005, das letzte in $C00007 abgespeichert.

Ist das Datenregister der Zieloperand des Befehls, so wird die ganze Operation umgekehrt ausgeführt. MOVEP muß stets mit Bedacht gehandhabt werden, da er sich vom normalen MOVE doch in mancher Hinsicht unterscheidet. Zum einen werden bei seiner Ausführung keine Condition-Codes verändert, und zum anderen können nur alternierende Bytes behandelt werden. Der Zugriff auf diese erfolgt nicht schneller als sonst auch, und so ist oft der MOVE-Befehl im Byteformat einfacher anzuwenden.

Vorteilhaft ist MOVEP nur dann einzusetzen, wenn große Datenmengen zu übertragen sind. Wird beispielsweise ein für 8-Bit-Prozessoren entwickelter Arithmetik-Prozessor an den 68000 angeschlossen, so kann ein 32-Bit-Wert auf einen Schlag übertragen werden und erspart so 4 MOVEs und drei Shiftoperationen.

4
Stacks und Unterprogramme

In einem Computer werden die von einem Programm verwendeten Daten häufig in sogenannten **Stacks** (Stapeln) abgespeichert. Bei dieser Methode werden die zu speichernden Daten quasi übereinander gestapelt, einem Papierstapel oder Holzstoß nicht unähnlich. Dabei ist immer nur der »oberste«, d.h. der zuletzt abgelegte Wert des Stacks zugreifbar; er kann vom Stack entfernt werden, worauf der nächste Wert zugreifbar wird. Man spricht daher vom Ablegen »auf« den bzw. Entfernen »vom« Stack.

Der Stack wird im Computer durch einen Speicherbereich realisiert, der von oben nach unten adressiert wird. Wenn wir daher bei dieser »umgedrehten« Methode vom obersten Stackplatz sprechen, ist die Speicherstelle mit der kleinsten verwendeten Adresse gemeint.

Zu Beginn wird ein Register auf die höchste Adresse des Stackbereiches gesetzt. Soll ein Wert gespeichert werden, so wird der Inhalt dieses Registers um die Länge (in Byte) des zu speichernden Wertes vermindert und dieser an der durch das nun veränderte Register adressierten Adresse abgelegt. Der Inhalt des Registers (und oft auch das Register selbst) wird daher auch als Stackzeiger bezeichnet. Bei weiteren abzulegenden Daten wird analog verfahren, so daß die Daten auf dem Stack hintereinander abgespeichert werden.

Die in einem Stack abgespeicherten Daten können nur in der umgekehrten Reihenfolge wieder ausgelesen werden. Diese Eigenschaft wird mit **LIFO** (last in, first out) bezeichnet. Um Daten vom Stack zu entfernen, wird zunächst der Wert von der durch den Stackzeiger adressierten Stelle gelesen und anschließend der Stackzeiger um die Größe des gelesenen Objekts erhöht.

Alle acht Adreßregister des 68000 können als Stackzeiger verwendet werden. Das Erhöhen und Erniedrigen des Zeigers wird durch Einsatz von Postincrement- und Predecrementmodus realisiert, so daß nur ein Register auf den Stackanfang (höchste Adresse des Stacks) gesetzt werden muß.

Stacks werden beispielsweise verwendet, wenn bei einer Berechnung Register benötigt werden, deren ursprüngliche Inhalte nach der Ausführung der Berechnung erhalten bleiben sollen:

```
          MOVEA.L  #$2000,A3  Setzt A3 auf den Stackanfang

* Die Datenregister werden mit wichtigen Werten belegt

           ...
          MOVE.L   D0,-(A3)   Rettet D0 auf dem Stack
          MOVE.L   D1,-(A3)   Rettet D1 auf dem Stack
XXX       MOVE.L   #123,D0    D0 und D1 werden nun verändert
           ...
          MOVE.L   (A3)+,D1   Holt den alten Inhalt von D1 zurück
          MOVE.L   (A3)+,D0   Holt den alten Inhalt von D0 zurück
           ...                Die alten Werte von D0 und D1
           ...                werden weiter verwendet
```

In diesem Beispiel wird zunächst A3 als Stackzeiger initialisiert. Im Zuge der Berechnung werden alle Register mit wichtigen Werten belegt. Bei der Ausführung der ab dem Label XXX stehenden Befehle werden weitere Register benötigt, wobei jedoch alle momentanen Registerinhalte später im Programm noch von Bedeutung sind.

Eine Möglichkeit wäre nun, die Inhalte der benötigten Register in reservierten Speicherbereichen abzulegen und sie bei Bedarf wieder zu lesen. Dieses Verfahren hat allerdings mehrere Nachteile: Zum einen werden so Speicherplätze nur für diesen Zweck reserviert, und zum anderen können z.B. bei rekursiven Aufrufen eines Unterprogrammes diese Speicherplätze überschrieben werden. Einfacher und sicherer ist hier die Verwendung eines Stacks. Auch das Schreiben reinen und positionsunabhängigen Codes wird so erleichtert.

Die aktuellen Werte der Datenregister werden also auf dem Stack gespeichert. Anfangs hatte A3 den Inhalt $2000; durch Verwendung des Predecrementmodus bei dem MOVE-Befehl im Vollwortformat wird davon zunächst vier abgezogen. Der Inhalt von D0 wird an der durch den neuen Wert des Stackzeigers adressierten Stelle abgelegt, oder anders ausgedrückt, D0 wird in den Speicherstellen $1FFC bis $1FFF gespeichert. Der nächste Befehl speichert D1 in den Adressen $1FF8 bis $1FFB; A3 enthält letztlich den Wert $1FF8.

D0 und D1 können nun beliebig verwendet werden. Nach Beendigung der Teilberechnung werden die alten Werte von der durch A3 adressierten Stelle wieder geladen. Durch Verwendung des Postincrementmodus werden also D1 von $1FF8 und D0 von $1FFC geholt. Dabei darf jedoch nie vergessen werden, daß die Daten in umgekehrter Reihenfolge wieder vom Stack zu entfernen sind.

Zur Behandlung von Stacks gibt es eine spezielle Version des MOVE-Befehls. Im obigen Beispiel haben wir zum Abspeichern zweier Register zwei Befehle verwendet. Zum Retten aller 16 Register wären somit 16 Befehle, und damit 32 Byte Code und entsprechend Zeit nötig.

Der Befehl MOVEM dient zum Speichern von Registern auf dem Stack. Bei seiner Ausführung werden bis zu 16 Register auf einmal auf den Stack geschrieben oder von ihm geladen. Als ein Operand findet dabei eine Liste von Registern Verwendung; der andere ist eine Operandenadresse. Die Registerliste wird in ein Wort konvertiert, wobei für jedes aufgeführte Register das entsprechende Bit gesetzt und für jedes nicht aufgeführte das entsprechende Bit gelöscht wird. Alle 16 Register können somit durch Verwendung eines nur 4 Byte langen Befehls gespeichert oder geladen werden, wobei die Ausführung dieses Befehls auch deutlich weniger Zeit benötigt als die entsprechende Anzahl expliziter MOVEs.

Unser Beispiel kann daher wie folgt abgeändert werden:

```
          MOVEA.L  #$2000,A3  Setzt A3 auf den Stackanfang

* Die Datenregister werden mit wichtigen Werten belegt

            ...
          MOVEM.L  D0-D1,-(A3)     Rettet D0 und D1 auf dem Stack
XXX       MOVE.L   #123,D0         D0 und D1 werden nun verändert
            ...
          MOVEM.L  (A3)+,D0-D1     Holt die alten Inhalte von
*                                  D0 und D1 zurück
           ...                     Die alten Werte von D0 und D1
           ...                     werden weiter verwendet
```

Die Registerliste wird durch Angabe des ersten und des letzten zu rettenden Registers, getrennt durch einen Bindestrich, spezifiziert. Eine weitere Möglichkeit besteht in der Angabe eines Registers, gefolgt von einem Schrägstrich, welcher wiederum von einem weiteren Registernamen gefolgt wird. Beide Möglichkeiten können auch iteriert und kombiniert werden:

```
MOVEM.L  D5-D7,D3/A0-A2/A6,-(A3)
```

rettet D5, D6, D7, D3, A0, A1, A2 und A6.

Einige Assembler verbieten die gemeinsame Aufzählung von Daten- und Adreßregistern, so daß

```
MOVEM.L  D0-D7/A0-A6,-(A7)
```

zum Retten aller Register außer A7 auf den durch A7 adressierten Stack angegeben werden muß.

Die Reihenfolge der Register ist dabei unabhängig von deren Position in der Liste, da der Assembler nur die den einzelnen Registern entsprechenden Bits setzt. Die Register werden in der Reihenfolge A7 - A0, D7 - D0 gerettet und in der umgekehrten Folge wieder vom Stack geladen.

Bei den bisher aufgeführten Beispielen haben wir zum Retten von Werten stets den Predecrement- und zum Laden den Postincrementmodus verwendet. Dabei werden die Register wie angegeben abgespeichert und gelesen, und das als Stackzeiger benutzte Adreßregister wird entsprechend der Gesamtgröße der bewegten Register erhöht bzw. erniedrigt. Bei dieser Anwendung verhält sich der MOVEM-Befehl fast wie die entsprechende Anzahl von MOVE-Befehlen, obwohl es einige Unterschiede gibt.

Zunächst einmal kann MOVEM nur mit Wort- oder Vollwortformat benutzt werden; einzelne Bytes können mit diesem Befehl nicht auf dem Stack abgelegt werden. Werden Wortoperanden vom Stack geholt, so wird das ganze Register verändert; die oberen 16 Bits werden dabei durch Sign-Extension gesetzt. Daher können die unteren 16 Bits eines Registers nicht auf dem Stack gerettet, in einer Berechnung geändert und anschließend wiederhergestellt werden, ohne dabei die oberen 16 Bits zu beeinflussen. Normalerweise sollte daher das Vollwortformat des Befehls verwendet und der gesamte Registerinhalt gerettet werden.

Des weiteren verwendet MOVEM die sogenannte **'pre-fetch'**-Technik. Dies bedeutet, daß der 68000, um die Register so schnell wie möglich zu übertragen, Speicherstellen bereits kurz vor dem Zeitpunkt liest, zu dem sie eigentlich benötigt werden. Dadurch wird die Übertragung beschleunigt, jedoch hat der Prozessor nach Abarbeitung des Befehls ein Speicherwort zuviel gelesen. Normalerweise macht dies nichts aus, da alle Registerinhalte stimmen und ein gelesenes und anschließend vergessenes Wort niemanden stört. Beginnt der Stack jedoch an der höchsten verfügbaren Speicherstelle, so versucht der Prozessor beim Entfernen der Registerinhalte vom Stack ein Wort außerhalb des vorhandenen Speicherbereiches zu lesen. Zwar wird dieses Wort anschließend vergessen, jedoch verursacht der Zugriff auf diese nicht existente Adresse normalerweise einen Busfehler; daher wird das Programm nicht in der erwarteten Weise abgearbeitet.

Schließlich setzt der MOVE-Befehl die Condition-Codes, MOVEM hingegen nicht. Dies ermöglicht es, in einem Unterprogramm ein Status-Bit zu setzen, um dem rufenden Programm das korrekte Abarbeiten zu signalisieren, wobei dieser 'Return-Code' durch das Wiederherstellen der Register nicht verändert wird.

Wie bereits erwähnt, werden die Register nur dann in der beschriebenen Form gerettet und geladen, wenn Predecrement- und Postincrementmodus verwendet werden. Der Befehl MOVEM kann aber auch mit anderen Adressierungsarten benutzt werden, wobei diese vom Typ 'control' sein müssen. Bei einem Transfer in den Speicher muß dieser 'control alterable' adressiert sein, so daß relativ zum Programmzähler also nur zum Lesen aus dem Speicher adressiert werden kann.

Die Reihenfolge der Übertragung der Register lautet bei diesen anderen Adressierungsarten: D0-D7, A0-A7, also genau wie bei Verwendung des Postincrementmodus.

```
MOVEM.L   D0-D7,$2000
```

speichert daher den Inhalt von D0 in den Speicherstellen $2000 bis $2003, den von D1 in $2004 bis $2007 usw. Um die Register wieder zu laden, kann folgender Befehl verwendet werden:

```
MOVEM.L   $2000,D0-D7
```

4.1 UNTERPROGRAMME

In einem der vorigen Beispiele zu MOVEM verwendeten wir A7 als Register für den Stackzeiger. Obwohl jedes andere Adreßregister ebenso gut dafür geeignet wäre, wird dennoch meistens A7 benutzt, da einige Befehle den Inhalt von A7 automatisch als Zeiger auf einen Stack interpretieren. Es gibt in Wirklichkeit zwei Ausführungen von A7: den Benutzer-Stackzeiger (USP = user stack pointer) und den Supervisor-Stackzeiger (SSP). Im folgenden ist A7 stets der SSP.

Beim Schreiben eines Programmes für den 68000 sollte man sich vergewissern, daß der Stackzeiger auch wirklich auf den Anfang des Stacks zeigt. Meistens wird dies bereits vom Betriebssystem oder vom Monitor durchgeführt, kann aber auch vom Programm selbst vorgenommen werden:

```
MOVE.L    $2000,A7
```

definiert den Bereich unterhalb von $2000 als Stack. Werden Stackbereiche überschrieben, kann ein gewaltiges Chaos entstehen; daher sollte für das Wachstum des Stacks eine mehr als großzügige Grenze vorgesehen werden. Unser Stack kann nun von $2000 an abwärts wachsen, und wir können von $2000 an aufwärts unser Programm ablegen, da der ursprüngliche Stackanfang nicht verwendet wird, wenn der Stack im Predecrementmodus beschrieben wird.

Nachdem wir nun unseren Stackzeiger initialisiert haben, können wir einige Befehle einführen, die den Inhalt von A7 als Zeiger auf einen Stack interpretieren. Am ehesten ist dies wohl bei dem Befehl BSR (branch to subroutine) zu sehen. Dieser sehr wichtige Befehl veranlaßt wie BRA einen Sprung zu einer anderen Stelle im Programm. Anders als bei BRA ist es bei Verwendung von BSR möglich, am Sprungziel die Stelle zu ermitteln, von wo der Sprung ausging.

Der Befehl

```
BRA       ERROR
```

zum Beispiel könnte an mehreren Stellen im Programm auftreten. Beim Label ERROR sollte dann eine Fehlermeldung ausgegeben und das Programm angehalten werden. Das

bei ERROR beginnende Programmstück kann jedoch nicht herausfinden, von woher der Sprung erfolgte. Um eine passende Fehlermeldung ausgeben zu können, müßte daher vorher ein Wert in einem Register oder einer Speicherzelle abgelegt werden, der Angaben über die Fehlerursache enthält. Wir könnten aber auch nur die Fehlermeldung ausgeben und anschließend mit der Programmausführung fortfahren wollen. In diesem Fall ist der Befehl BSR eher angebracht. Bei seiner Ausführung rettet der Prozessor zunächst die Adresse des nächsten Befehls und springt dann zur angegebenen Adresse. Der dort vorliegende Code kann diese gerettete Adresse zur Rückkehr verwenden und nach seiner Abarbeitung mit dem nächsten Befehl hinter dem Sprung fortfahren. Weiterhin kann diese Adresse verwendet werden, um die Fehlerursache festzustellen.

Die bei der Ausführung von BSR gerettete Adresse wird als **Rücksprungadresse** (return address) bezeichnet, und sicher haben Sie schon erraten, daß sie auf dem durch A7 adressierten Stack abgelegt wird. Läge der Programmzähler in einem Register vor, sagen wir Register PC, so entspräche die Befehlsfolge

```
        MOVE.L   PC,-(A7)    Rücksprungadresse auf dem Stack
*                            retten
        BRA      ERROR       Sprung zum Unterprogramm
```

der Ausführung von

```
        BSR      ERROR
```

Es gibt allerdings keine Möglichkeit, den Inhalt des Programmzählers explizit als Operanden zu verwenden. Der BSR-Befehl vermindert A7 um vier, speichert die Rücksprungadresse in der durch den neuen Inhalt von A7 spezifizierten Speicherstelle ab und springt zu der angegebenen Adresse, und all dies in einem Zug.

Als nächstes benötigen wir eine Möglichkeit, diesen Wert vom Stack zu holen und ihn für den Sprung zurück hinter den Aufruf des Unterprogramms zu verwenden. Dieser Wert kann ggf. explizit vom Stack gelesen werden, so daß zum Rücksprung der folgende Programmteil verwendet werden könnte:

```
        MOVEA.L  (A7)+,A6    Holt die Rücksprungadresse
        JMP      (A6)        und springt dorthin
```

Der übliche Weg, aus einem Unterprogramm zurück zu springen, besteht aus dem Aufruf des Befehls RTS (return from subroutine). Dieser ist das genaue Gegenstück zu BSR. Er liest die Rücksprungadresse vom Stack, erhöht A7 um vier und springt zu der gelesenen Adresse.

Die sich bei Verwendung des Stacks ergebenden Vorteile liegen auf der Hand. BSR kann beliebig oft aufgerufen werden, sogar aus Programmbereichen, die selbst als Unterprogramme aufgerufen wurden. Der Stackzeiger wird dabei jedesmal um vier erhöht und die Rücksprungadresse auf den Stack geschrieben; jeder RTS-Befehl holt sich

die oberste Adresse vom Stack und springt an sie zurück, um mit der Ausführung des auf BSR folgenden Befehls fortzufahren.

Diese besondere Bedeutung von A7 darf nie vergessen werden, und ebensowenig, daß sein Inhalt auf einen als Stack zu verwendenden Speicherbereich zeigen muß. Einige Assembler verwenden daher den Namen SP (stack pointer) für A7. Die Verwendung dieses Namens hilft, sich dessen besonderer Bedeutung gewahr zu sein. In vielen Fällen kann A7 wie jedes andere Adreßregister verwendet werden. Wie bereits erwähnt gibt es jedoch einen wichtigen Unterschied: Auf dem Stack abgelegte Werte müssen stets an einer Wortgrenze ausgerichtet sein. Da dies hardwaremäßig vorgenommen wird, wird bei Verwendung von A7 als Operand in einem Befehl mit Byteformat und Predecrement- oder Postincrementmodus der Registerinhalt von A7 anders als bei den anderen Registern nicht um eins, sondern um zwei verändert.

Das Konzept der Unterprogramme ist sehr bedeutend; es dürfte jedem mit etwas Programmiererfahrung bekannt sein. Anstatt Programmstücke zur Ausführung ein und derselben Operation mehrfach zu schreiben, werden sie nur einmal geschrieben und als Unterprogramm verwendet. Ein Beispiel dafür ist der zur Ausgabe eines Zeichens benötigte Code, wie er in Kapitel 3 beschrieben wurde. Es wäre eine gewaltige Platzverschwendung, in einem Programm vor jeder Übergabe eines Zeichens an einen ACIA zur Prüfung dessen Zustands (bereit / beschäftigt) lange Codestücke einzufügen. Stattdessen schreiben wir ein Unterprogramm, um ein Zeichen auszugeben, und rufen dieses dann mit BSR auf. Nach Ausgabe des Zeichens kommen wir dann mittels RTS wieder in unsere Berechnung zurück.

```
WRCH    BTST    #1,A_CTRL
        BEQ.S   WRCH
        MOVE.B  D0,A_DATA
        RTS
```

Dieses Unterprogramm gibt ein Zeichen auf einem an eine serielle Leitung angeschlossenen Gerät aus, sofern diese Leitung ordnungsgemäß initialisiert wurde. Wir müssen nur das Zeichen im unteren Byte von D0 ablegen und einen BSR zum Label WRCH (write character) ausführen, um dieses Unterprogramm zu benutzen. Das Unterprogramm wartet, bis der ACIA Bereitschaft signalisiert, und schreibt dann das Zeichen in das Datenregister des Bausteins. Der Programmablauf wird anschließend bei dem auf den Sprung folgenden Befehl wieder aufgenommen.

Normalerweise hat jeder Programmierer eine Sammlung gebrauchsfertiger Unterprogramme zur Hand. Fast alle Programme haben gewisse Ähnlichkeiten. So besteht z.B. bei beinahe jedem Programm die Notwendigkeit, irgendwelche Ergebnisse auszugeben. Daher sollte ein Programm soweit wie möglich in Unterprogramme gegliedert werden, um bestimmte Teile später in anderen Programmen wieder einsetzen zu können. Eine normale Unterprogrammsammlung besteht aus einer ganzen Anzahl verschiedener Unterprogramme. Allein zur Ausgabe gibt es viele sinnvolle Operationen wie

das Ausgeben eines Zeichens, einer Dezimal- oder Hexadezimalzahl, eines kompletten Textes, eines Speicherauszugs usw.

Diese Unterprogrammsammlung wächst im Laufe der Zeit; einige Programmiertechniken sollten jedoch immer beachtet werden. Diese sind unabhänhig von der verwendeten Programmiersprache und dem Prozessor. Zu unserem Glück unterstützt der Befehlssatz des 68000 die strukturierte Programmierung sehr.

Als erstes sollte beachtet werden, daß ein Unterprogramm unter allen Bedingungen verwendbar ist. Als Beispiel wollen wir unserer Sammlung ein Unterprogramm zur Ausgabe eines Textes hinzufügen. Bei Aufruf soll in A4 die Adresse des Textes stehen. Dieser besteht aus einer Folge von Zeichen, die durch ein Byte mit Wert Null begrenzt ist. Dieses Unterprogramm kann dann wie folgt aussehen:

```
WRITES   MOVE.B   (A4)+,D0       Holt ein Zeichen des Textes
         BEQ.S    WOVER          Ende des Textes erreicht
         BSR.S    WRCH           Ausgabe des Zeichens
         BRA.S    WRITES         Nächstes Zeichen
WOVER    RTS                     Rücksprung
```

Wir holen uns hier zunächst ein Zeichen (Byte) des Textes in das Register D0, wobei der Zeiger auf den Text entsprechend weitergesetzt wird. Hatte das Zeichen den Wert Null, so springen wir zum Label WOVER und von dort zurück in das rufende Programmstück. Anderenfalls geben wir das Zeichen durch Aufruf des ebenfalls in unserer Sammlung stehenden Unterprogramms WRCH aus und beginnen die Schleife von vorne. Wir haben hier bei allen Sprüngen die kurze Form des Befehls gewählt, mit der nur Sprünge bis zu 128 Byte weit möglich sind. Wir müssen daher gewährleisten, daß das Label WRCH nicht zu weit von WRITES entfernt liegt.

Obwohl dieses Unterprogramm unseren Anforderungen entspricht, wirft es eine Anzahl Probleme auf. Zunächst einmal müssen wir bei jedem Aufruf von WRITES daran denken, daß der Inhalt von D0 verändert wird, da das untere Byte immer auf Null gesetzt wird. Im Augenblick mag es uns egal sein, was mit D0 passiert, aber für ein in allen Situationen verwendbares Unterprogramm ist das Verändern von Registern eine denkbar ungünstige Eigenschaft.

Sehen wir uns das Programmstück etwas genauer an, so stellen wir fest, daß auch das Register A4 seinen Wert ändert. Es wird so gesetzt, daß es auf das erste Byte hinter dem Text zeigt. Obwohl der Benutzer unseres Unterprogramms wissen muß, daß die Adresse des Textes in A4 übergaben werden muß, ist es unfair von uns, als Seiteneffekt der Ausgabe das Register A4 zu verändern. Unterprogramme sollten daher nie die Inhalte von Registern ändern, außer in den Fällen, wo sie die Ergebnisse ihrer Berechnung in Registern an das rufende Programm zurückgeben. Nachdem wir uns nun bereits entschieden haben, die Inhalte der Register in einem Unterprogramm nicht zu ändern, müssen wir einen Weg finden, diese Werte zwischenzuspeichern. Eine Möglichkeit be-

stände darin, einfach einen Speichernbereich zu reservieren und als Zwischenspeichern zu verwenden. Dies scheint zwar zu funktionieren, bringt aber wieder andere Probleme mit sich. Zum einen müßten wir für jedes Unterprogramm einen eigenen Speicherbereich verwenden, da sonst beim Aufruf weiterer Unterprogramme die geretteten Werte evtl. wieder überschrieben werden könnten. Diese Zuweisung eigenen Speichers an jedes Unterprogramm verschwendet Platz und ist nicht einfach zu organisieren. Des weiteren wären diese Unterprogramme nicht re-entrant (siehe Kap. 2).

Die bessere Methode besteht darin, die Registerinhalte mittels MOVEM auf den Stack zu retten. Um BSR verwenden zu können, mußten wir bereits ein Register mit der Adresse eines passenden Speicherbereiches laden. Solange dieser Bereich auch bei der maximalen Verschachtelungstiefe noch groß genug für alle Rücksprungadressen und geretteten Register ist, sind wir fein raus. Besonders vorteilhaft ist dabei, daß wir den Speicherplatz effizient nutzen, indem wir nur tatsächlich benötigte Speicherplätze belegen. Wir können so Hunderte von Unterprogrammen verwenden, die alle Platz zum Retten der Register benötigen; die maximale Stackauslastung ist nur von der maximalen Anzahl von aktiven, d.h. aufgerufenen Unterprogrammen abhängig.

Unser verbessertes Unterprogramm zur Textausgabe sieht nun wie folgt aus:

```
WRITES   MOVEM.L  D0/A4,-(SP)    Register retten
WR1      MOVE.B   (A4)+,D0       Zeichen aus dem Text holen
         BEQ.S    WR2            Ende des Textes erreicht
         BSR.S    WRCH           Ausgabe des Zeichens
         BRA.S    WR1            Nächstes Zeichen
WR2      MOVEM.L  (SP)+,D0/A4    Wiederherstellen der Register
         RTS                     Rücksprung
```

Nunmehr können wir WRITES beliebig aufrufen und uns dabei sicher sein, daß kein Register verändert wird. Außer dem Bereich unterhalb des Stackzeigers wird keine Speicherstelle verändert, und der Zustand dieses Bereiches ist nach Unterprogrammaufrufen sowieso uninteressant.

Zwei weitere Regeln sollten bei der Entwicklung von allgemein verwendbaren Unterprogrammen beachtet werden. Die erste besagt, daß die an ein Unterprogramm übergebenen Argumente sowie die Ergebnisse dieses Unterprogramms stets in denselben Registern übergeben werden sollten. Man könnte beispielsweise die Argumente in den Registern D1, D2 usw. übergeben und ein einzelnes Ergebnis in D0 zurückgeben. Leider ist dies nicht immer durchführbar, u.a. weil manche Unterprogramme ihre Argumente in Daten- und andere diese in Adreßregistern benötigen; dennoch sollte diese Regel soweit wie möglich beachtet werden. Für den Programmierer ist es so einfacher, vor dem Aufruf eines Unterprogramms die richtigen Werte in die richtigen Register zu laden.

Auch ist es wichtig, daß in einem Unterprogramm nur an einer Stelle ein Rücksprung mittels RTS erfolgt. Es ist sinnvoller, anstelle mehrerer RTS-Befehle nur

einen zu verwenden und diesen ggf. anzuspringen. So kann das Wiederherstellen von Registern und das Freigeben von Stackbereichen an einer einzigen Stelle im Unterprogramm erfolgen. Wird bei einer Erweiterung des Unterprogramms ein weiteres Register verwendet, so muß dieses gerettet und wiederhergestellt werden. Das Unterprogramm muß dazu dann nur an einem einzigen Ein- bzw. Aussprungpunkt entsprechend angepaßt werden. Ein häufig auftretender Programmierfehler besteht darin, ein Register auf dem Stack zu retten und zu vergessen, es vor einem RTS wieder zu entfernen. Die Auswirkungen dieses Fehlers sind meist katastrophal, da das Programm versucht, zu einer durch den Inhalt des geretteten Registers angegebenen Stelle zu springen. Dieser Fehler ist nur sehr schwer zu entdecken, da bei der Fehlersuche nur der neue Programmzähler angegeben wird, nicht aber, wie er zustande kam.

4.2 ABSOLUTE SPRÜNGE

Natürlich muß man in der Lage sein, zu einem anderen Teil des Programms springen zu können. Bisher kennen wir bereits Befehle wie BRA, mit dem wir Schleifen aufbauen können, BSR zum Sprung in ein Unterprogramm oder BEQ als Beispiel für einen bedingten Sprung.

In all diesen Fällen impliziert der Name des Befehls eher eine Verzweigung (branch) als einen Sprung. Der Grund dafür ist darin zu sehen, daß bei diesen Befehlen das Sprungziel eine Adresse relativ zur momentanen Position ist. Auch wenn wir einfach

```
BRA     LOOP
```

schreiben und damit die Adresse des Labels LOOP meinen, bestimmt der Assembler die Differenz zwischen dem augenblicklichen Inhalt des Programmzählers und der durch das Label LOOP spezifizierten Adresse. Diese Differenz wird im Code für diesen Befehl als Operand eingetragen. Diese Differenz (auch Offset genannt) kann positiv oder negativ sein, je nachdem, ob der Sprung vorwärts oder rückwärts erfolgte, und sie kann als Wort oder - bei Verwendung der Kurzformats des Befehls - als Byte angegeben sein.

Alternativ zu BRA kann der Befehl JMP verwendet werden. Die im JMP-Befehl enthaltene Adresse ist die tatsächliche Adresse, an welcher der Programmablauf nach dem Sprung aufgenommen wird. Die Auswirkungen des Befehls

```
JMP     LOOP
```

sind denen des entsprechenden BRA-Befehls ähnlich, jedoch gibt es eine Anzahl Unterschiede.

Zunächst einmal benötigt der JMP-Befehl mehr Platz, da die Adresse vollständig im Befehlscode untergebracht wird.

Zum zweiten ist ein Programmstück bei Verwendung von absoluten Sprüngen zu eigenen Labeln nicht mehr relokatibel. Viele Befehle des 68000 erlauben es aber, positionsunabhängigen Code zu erzeugen. Zum Beispiel kann durch einen BRABefehl spezifiziert werden, zur 24 Byte weiter vorne liegenden Adresse zu springen. Diese Anweisung ist absolut unabhängig von der tatsächlichen Position des Programmes im Speicher. Der Operand eines JMP-Befehls kann in jeder der Adressierungsarten vom Typ 'control' spezifiziert werden. Bei Angabe eines absoluten Wertes ist das Programm dann natürlich nur lauffähig, wenn es an die durch ORG angegebene Adresse geladen wurde, da bei der Verwendung absoluter Adressen eine ganz bestimmte einzelne Speicherstelle definiert wird.

Meistens ist nichts dabei, ein Programm an eine fest vorgegebene Stelle im Speicher zu laden. Die Verwendung eines Betriebssytems wird dadurch jedoch so gut wie unmöglich, da dieses ein Programm an eine beliebige freie Stelle lädt. Ist das Programm relokatibel, so läuft es anschließend ohne jede Änderung ab. Anderenfalls muß der Programm-Code **Relokations-Information** enthalten. Diese wird von einigen Assemblern erzeugt und erlaubt es dem Betriebssystem, nach dem Laden des Programmes die in den Befehlen enthaltenen absoluten Adressen der tatsächlichen Lage anzupassen. Dadurch wird die Lauffähigkeit des Programms bei jeder Ladeadresse sichergestellt. Leider erzeugen nicht alle Assembler brauchbare Relokations-Information, und daher ist es prinzipiell günstiger (auch zur Entlastung des Betriebssystems), relokatiblen Code zu generieren.

Ein weiterer Vorteil von relokatiblem Code besteht darin, daß dieser nicht nur an eine beliebige Stelle geladen, sondern bei Bedarf auch noch im Speicher verschoben werden kann. Dabei muß allerdings beachtet werden, daß das Programm nicht während des Laufs beeinflußt wird, da die auf dem Stack vorliegenden Rücksprungadressen absolute Speicherzellen bezeichnen. Eine Verschiebung kommt daher nur zwischen einzelnen Programmläufen in Frage.

Das alles soll nun aber nicht bedeuten, daß JMP in relokatiblen Programmen nicht sinnvoll einsetzbar ist. Ganz im Gegenteil bewirkt der JMP-Befehl nur dann eine Positionsabhängigkeit, wenn er zum Sprung im eigenen Programmbereich verwendet wird. Unbedenklich ist seine Verwendung zum Sprung in andere Bereiche, deren Adressen festliegen. So kann etwa ein Monitor in einem **EPROM** an einer festen Stelle liegen, und dieser Monitor eine sogenannte Warmstartadresse besitzen, die von einem Programm nach seiner Abarbeitung angesprungen werden muß. Ein Programm besitzt dann

```
JMP     WARMS     Warmstart
```

als letzten Befehl. Dadurch wird dann das System wieder in einen definierten Zustand versetzt, unabhängig vom Ablauf und der Lage unseres Programmes.

Auch der Befehl BSR hat ein Gegenstück: JSR (jump subroutine). Wie JMP verwendet JSR keinen Offset, sondern eine absolute Adresse. Dieser Befehl dient zum Aufruf von Unterprogrammen, deren absolute Adresse feststeht; insbesondere sind dies Monitor- oder Betriebssystem-Routinen.

JMP und JSR sind sehr wichtige Befehle, vor allem, da sie jede Adressierungsart vom Typ 'control' zulassen. Im obigen Beispiel haben wir absolut adressiert, um zu einer festen Speicherstelle zu springen. In unserem eigenen Programm können wir aber auch relativ zum Programmzähler adressieren und dabei die Positionsunabhängkeit beibehalten. Die Wirkung der Befehle in diesem Fall ist dann der von BRA bzw. BSR sehr ähnlich.

Die vielseitigste Adressierungsart hier ist wohl eine aus der Gruppe der indizierten Adressierungsmodi. Nehmen wir einmal an, daß wir ein Programm schreiben wollen, das auf einzelne eingegebene Zeichen reagiert. Bei jeder Eingabe eines Zeichens wird ein Unterprogramm angesprungen, das diese Aufgabe erledigt. Wir können dies programmieren, indem wir eine Tabelle verwenden, in der jedem Zeichen ein Vollwort zugeordnet ist. Jeder Eintrag in dieser Tabelle enthält dann die Adresse eines Unterprogramms, das bei Eingabe des entsprechenden Zeichens aufgerufen werden soll. Die Adresse des bei Eingabe des Zeichens 'A' (hex 41) anzuspringenden Unterprogramms liegt dann an der Position 65, d.h. ab Byte 260 in dieser Tabelle.

```
ASL.L    #2,D0          Multiplikation mit 4
MOVEA.L  #TABELLE,A3    Adresse der Tabelle
JSR      0(A3,D0.W)     Sprung ins Unterprogramm
```

In der ersten Zeile wird der Inhalt von D0 mit vier multipliziert, indem er zweimal nach links geshiftet wird (siehe Kapitel 6). Dies ist nötig, da jeder Eintrag in der Tabelle vier Bytes benötigt. In der zweiten Zeile wird die Anfangsadresse unserer Sprungtabelle nach A3 geladen. In der letzten Zeile wird dann der Sprung ausgeführt, wobei sich das Sprungziel aus der Summe der Inhalte von A3 und D0 ergibt. Die Rücksprungadresse wird dabei automatisch auf dem durch A7 adressierten Stack abgelegt.

4.3 OPERANDENADRESSEN

Um die Adresse der Tabelle in das Register A3 zu laden, benutzten wir im vorigen Beispiel den Befehl MOVEA mit einem Direktoperanden. Dies geht zwar, aber der sich so ergebende Code ist dann positionsabhängig, da der Befehl letztlich nur einen festen Wert in das Register lädt, obwohl wir mit dem Label TABELLE etwas anderes assoziieren. Wird das Programm nicht an der durch ORG angegebenen Stelle geladen und zum Ablauf gebracht, so wird bei Ausführung des MOVEA-Befehls nicht die Adresse des Unterprogramms, sondern irgendein unsinniger Wert aus dem Speicher gelesen, der trotzdem für das Sprungziel gehalten wird. Die Auswirkungen des sich anschließenden Sprungs sind dann unvorhersehbar.

Als Ausweg aus dieser Misere bietet sich der Befehl LEA (load effective address) an. Als Zieloperanden benötigt dieser Befehl ein Adreßregister; die Adresse des Quelloperanden wird zunächst wie z.B. bei MOVE ausgewertet. Anstatt des an dieser Stelle liegenden Wertes wird nun jedoch die Adresse selbst in das Zielregister geladen.

Betrachten wir das folgende Beispiel:

```
ORG       $1000
LAB       DC.L      1234
MOVEA.L   LAB,A3
MOVEA.L   #LAB,A3
LEA.L     LAB.A3
```

Hier belegen wir zunächst vier Byte im Speicher ab Adresse $1000 mit dem Wert 1234. Der erste MOVEA-Befehl wertet zuerst die durch LAB spezifizierte Adresse ($1000) aus und lädt dann den Inhalt dieser Speicherstelle (1234) nach A3. Das zweite MOVEA lädt Register A3 mit dem durch LAB angegebenen Direktoperanden $1000. Dieser Befehl ist positionsabhängig und kann daher auch nur im absoluten Modus (ORG statt RORG) assembliert werden.

Der einfachere und sicherere Weg, um dies zu erreichen, besteht darin, die Adresse mittels LEA zu laden. Auch dieser Befehl bestimmt die durch LAB gegebene Adresse, verwendet dabei aber die Adressierungsart relativ zum Programmzähler, wodurch der sich ergebende Code relokatibel wird. LEA berechnet nur die Adresse und speichert sie in das angegebene Register, läßt jedoch den an dieser Adresse liegenden Wert unbeeinflußt. Dieser Befehl gibt natürlich nur dann Sinn, wenn er im Vollwortformat benutzt wird; LEA ist daher äquivalent zu LEA.L. Vergessen Sie nicht, immer die Länge eines Befehls mit anzugeben, da verschiedene Befehle verschiedene Längen als Defaultwert verwenden.

Der Befehl LEA ist recht bedeutend, da mit ihm z.B. im vorigen Beispiel der Verweis auf das Label LAB relativ zum Programmzähler gemacht werden kann, wodurch das Programm relokatibel wird. Um eine Adresse im eigenen Programm in ein Register zu laden, sollten Sie stets LEA verwenden. MOVE mit einem Direktoperanden sollte nur zum Laden von wirklich festen Werten benutzt werden.

LEA kann auch für kleinere Additionen verwendet werden, wenn diese Teil einer Adreßauswertung sind.

```
LEA       20(A3),A3
```

beispielsweise berechnet zunächst die durch 20(A3) spezifizierte Operandenadresse. Diese entspricht dem um zwanzig erhöhten Inhalt von A3; dieser Wert wird dann in das Register A3 geladen. Was dieser Befehl also tatsächlich bewirkt, ist, daß zum Inhalt von A3 zwanzig hinzuaddiert wird. Da jede Adressierungsart vom Typ 'control' erlaubt ist, wäre auch der Befehl

```
        LEA     20(A2,D1.L),A3
```

möglich. Dieser lädt A3 mit der Summe aus der Konstanten zwanzig und den Inhalten der Register A2 und D1. Obwohl Adressen normalerweise nur 24 Bit lang sind, werden hierdurch alle 32 Bits des Adreßregisters verändert.

Ein entfernter Verwandter von LEA ist der Befehl PEA (push effective address). Dieser Befehl wertet die im Quelloperanden angegebene Adresse genauso aus wie LEA, aber anstatt sie dann in ein Adreßregister zu laden, legt er sie auf dem Stack ab. Diese Operation ist auch ein Bestandteil der Befehle BSR und JSR, die bei der Ausführung die Adresse des nächsten Befehls auf den Stack retten. Die Wirkung des Befehls BSR ließe sich also folgendermaßen simulieren:

```
        PEA.L   NEXT        Rücksprungadresse auf dem Stack
*                           retten
        BRA     UNTERPR     Sprung zum Unterprogramm
NEXT    ...                 Rücksprungadresse
```

Wir speichern zunächst die Rücksprungadresse auf dem Stack, und verwenden zum Sprung in das Unterprogramm anstelle von BSR den BRA-Befehl. Wenn das Unterprogramm zum Rücksprung den Befehl RTS aufruft, wird dazu die auf dem Stack gerettete Adresse benutzt. Diese zeigt auf den mit NEXT bezeichneten Befehl, welcher der nächste nach dem Aufruf des Unterprogramms ist. Wir haben hier die Ausführung von BSR simuliert, jedoch muß die bei dem PEA-Befehl angegebene Adresse nicht unbedingt hinter den BRA-Befehl zeigen, sondern kann eine beliebige Stelle im Speicher adressieren.

Auch PEA kann zu einfachen Additionen verwendet werden.

```
PEA.L   20(A3)      Speichert A3+20 auf dem Stack
MOVE.L  (SP)+,D0    Lädt D0 mit dem geretteten Wert
```

Diese Befehlsfolge speichert den Inhalt von A3 plus zwanzig auf dem Stack, und liest anschließend diesen Wert vom Stack nach D0 zurück. Dies ist übrigens auch eine Möglichkeit, die Wirkung von LEA mit einem Datenregister als Zieloperanden zu erhalten.

4.4 ANFORDERUNG VON STACKBEREICHEN

In den vorigen Beispielen sahen wir, wie der Stack zum Speichern von Rücksprungadressen, Registern und Zwischenergebnissen verwendet werden kann. In all diesen Fällen mußten die gespeicherten Objekte in der umgekehrten Reihenfolge wieder vom Stack entfernt werden. Oft ist es aber auch sinnvoll, einen Speicherbereich zur Abspeicherung von Zwischenergebnissen anzufordern, der dann beliebig beschrieben und gelesen werden soll.

Eine Möglichkeit, dies zu erreichen, besteht darin, absolut adressierte Speicherplätze zu benutzen, aber dies führt wieder zu den Problemen mit der Positionsabhängigkeit; ein bestimmter, fester Speicherbereich wäre für unser Programm nötig. Auch könnten wir Platz im Programm selbst reservieren und diesen relativ zum Programmzähler adressieren. Dieser Methode wird jedoch vom Befehlssatz des 68000 ein Riegel vorgeschoben, da derart adressierte Speicherstellen nur gelesen werden können. So soll verhindert werden, daß sich Programme selbst verändern können.

Eine andere Lösung liegt in der Verwendung eines Adreßregisters, das auf einen zur Abspeicherung von Ergebnissen geeigneten Speicherbereich zeigt. Durch indirekte Adressierung über dieses Register mit Verwendung von Offsets können wir unsere Daten bequem abspeichern. Wenn A1 auf diesen Datenbereich zeigt, können wir wie folgt adressieren:

```
DATA1    EQU      0                Offset
DATA2    EQU      4
         ...                       Initialisierung von A1
         MOVE.L   #20,D1           Abzuspeichernder Wert
         MOVE.L   D1,DATA1(A1)     Speichern im Datenbereich
```

Diese Methode funktioniert zwar, liefert aber wiederum zwei neue Probleme. Das erste besteht darin, die oben durch Punkte symbolisierte Initialisierung von A1 vorzunehmen. Normalerweise müssen wir ein Unterprogramm des Betriebssystems aufrufen, um freien Bereich zu belegen und A1 als Zeiger zu initialisieren. Derartige Systemaufrufe können ziemlich zeitaufwendig sein; ein Beispiel einer verwendbaren Speicherzuweisungsroutine findet sich in Kapitel 6. Das zweite Problem besteht darin, daß wir diesen Systemaufruf jedesmal ausführen müssen, wenn wir ein Unterprogramm aufrufen, das einen eigenen Datenbereich benötigt. Nur so kann garantiert werden, daß jedes Unterprogramm von jedem anderen aufgerufen werden kann. Um zu verhindern, daß der gesamte freie Platz mit der Zeit von irgendwelchen Unterprogrammen regelrecht aufgefressen wird, muß der belegte Bereich beim Beenden eines Unterprogramms auch wieder freigegeben werden.

Die beste Lösung ist die, daß wir uns den für die Daten des Unterprogramms benötigten Platz vom Stack nehmen. Bis jetzt haben wir ein Wachstum des Stacks nur dann erlaubt, wenn tatsächlich irgendwelche Werte auf ihm abgelegt wurden. Ein Unterprogramm wurde aufgerufen, rettete ein Register und rief dann seinerseits ein Unterprogramm auf. Der Stack enthält zu diesem Zeitpunkt die Rückkehradresse für das zweite Unterprogramm, das gerettete Register und die Rücksprungadresse für das erste Unterprogramm.

Nun wollen wir einmal sehen, wie wir einen Bereich des Stacks als Datenbereich nutzen können. Bei Einsprung in unser Unterprogramm retten wir zunächst die Register, so daß auf dem Stack nun deren Werte und die Rücksprungadresse liegen. Nun initialisieren wir einen Zeiger auf den Datenbereich, sagen wir A1. Dieser enthält nun

den gleichen Wert wie der Stackzeiger A7. Anschließend verändern wir A7 derart, daß »unterhalb« von A1 genügend Platz für unsere Belange verfügbar ist. Bei Aufruf des zweiten Unterprogramms wird die Rücksprungadresse an der durch A7 adressierten Position im Stack gespeichert, außerhalb unseres Datenbereiches.

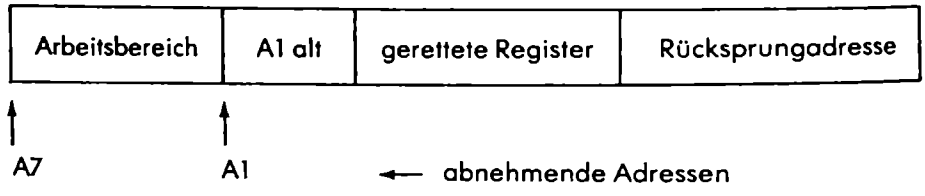
Das zweite Unterprogramm kann nun, falls nötig, ebenfalls Register retten und seinen eigenen Datenbereich vom Stack holen. Es muß aber auf jeden Fall den Inhalt von Register A1 retten, da dieser in der rufenden Prozedur noch benötigt wird. Auch der Inhalt des Stackzeigers muß vor dem Rücksprung unbedingt wiederhergestellt werden.

Dies hört sich vielleicht etwas kompliziert an, ist aber recht einfach, da der 68000 eine Anzahl spezieller Befehle zur Unterstützung dieser Methode bereitstellt. Bevor wir diese behandeln, wollen wir kurz rekapitulieren, was ein Unterprogramm zu tun hat, nachdem es betreten wurde und bevor es verlassen wird. Diese beiden Bereiche werden auch als Ein- und Aussprungfolgen bezeichnet.

Einsprungfolge:

1. Bei Einsprung zeigt A7 auf die Rücksprungadresse und A1 auf den vorherigen Datenbereich.
2. Alle benötigten Register werden gerettet. A7 zeigt auf die geretteten Register und die Rücksprungadresse; A1 immer noch auf den vorigen Datenbereich.
3. Retten des alten Inhalts von A1 auf den Stack. A1 wird nun mit A7 geladen, so daß es auf den neuen Arbeitsbereich zeigt. A7 wird entsprechend des benötigten Platzes erniedrigt; dabei muß unbedingt beachtet werden, daß der Stack von den hohen zu den niedrigen Adressen geht.

Auf dem Stack ergibt sich dann das folgende Bild:



Aussprungfolge:

1. Laden von A7 mit dem Inhalt von A1. Dadurch wird der Arbeitsbereich wieder freigegeben. A7 zeigt nun auf die geretteten Werte von A1, die der Arbeitsregister und auf die Rücksprungadresse. Zurückholen des alten Wertes von A1.
2. Wiederherstellen der Inhalte der Arbeitsregister vom Stack. A7 zeigt nun auf die Rücksprungadresse. Alle anderen Register haben nun wieder die gleichen Werte wie beim Einsprung (ausgenommen die Register, in denen evtl. Ergebnisse übergeben werden).
3. Holen der Rücksprungadresse vom Stack und Sprung.

Wir können uns nunmehr daran machen, all diese Anweisungen in Befehle des 68000 zu übersetzen. Stufe 1 der Einsprungfolge wird durch den Aufruf des Unterprogramms mittels JSR oder BSR realisiert. Stufe 2 wird durch einen MOVE- oder besser noch durch einen MOVEM-Befehl realisiert, der den Stack über A7 im Predecrementmodus adressiert. Stufe 3 wird in einem Zug durch Ausführung des Befehls LINK ausgeführt. Dieser speichert das als Quelloperand spezifizierte Adreßregister auf dem Stack und lädt es dann mit dem (neuen) Inhalt von A7. Abschließend addiert er den als Zieloperanden angegebenen Direktoperanden zum Stackzeiger A7. Da der Stack nach unten wächst, müssen wir im folgenden negative Werte als Displacement bei der Adressierung in unserem Datenbereich benutzen.

Nachdem wir nun in unserem eigentlichen Unterprogramm sind, können wir negative Offsets zu A1 als unsere Speicherplätze für beliebige Daten betrachten, solange wir nicht mit irgendwelchen Offsets unter den aktuellen Wert von A7 gelangen. Es ist daher wichtig, daß durch Angabe eines ausreichenden Displacements als Zieloperanden von LINK genügend Speicherplatz reserviert wird. Wird mit Offsets unterhalb des momentanen Werts von A7 adressiert, so können die dort abgelegten Werte durch den Aufruf weiterer Unterprogramme zerstört werden.

Die Aussprungfolge ist genauso einfach. Stufe 1 wird durch UNLK, dem exakten Gegenstück zu LINK, realisiert. Der Stackzeiger A7 wird mit dem Inhalt des als Argument angegebenen Registers geladen, und dann wird dieses Register mit dem nunmehr oben auf dem Stack liegenden Vollwort geladen. Stufe 2 ist wiederum ein MOVE oder MOVEM im Postincrementmodus, und Stufe 3 ist ein einfacher RTS.

```
* Standard-Einsprungfolge

          MOVEM.L   D0-D7/A0,-(SP)      Retten der Arbeitsregister
          LINK      A1,#-32             Reservierung von 8
*                                       Vollworten
          ....
* Abarbeitung des Unterprogramms. Dabei werden -4(A1) bis -32(A1)
* für Daten verwendet. Der Aufruf anderer Unterprogramme ist
* uneingeschränkt möglich.
          ....
* Standard-Aussprungfolge

          UNLK      A1                  Freigabe des Arbeitsspeichers
          MOVEM.L   (SP)+,D0-D7/A0      Wiederherstellung der Register
          RTS                           Rücksprung
```

Diese Arrangement wird häufig bei der Implementation höherer Programmiersprache wie **Pascal** oder **Ada** angetroffen. Wenn Sie ein Unterprogramm schreiben, das von Programmen einer höheren Programmiersprache aufgerufen werden soll, so müssen Sie auch bei Ihrem Unterprogramm auf Einhaltung des Standards für Unterprogramme dieser Programmiersprache achten. In verschiedenen Implementationen werden sich unvermeidlich Unterschiede zwischen den tatsächlich verwendeten Methoden ergeben.

4.5 EIN SPEICHERTEST-PROGRAMM

Wir haben nun bereits genug über den 68000 gelernt, um ein vollständiges Programm schreiben zu können. Dieses ist zwar nicht besonders aufregend, enthält aber Beispiele für die bisher kennengelernten Befehle.

Dieses Programm soll die Speicherbereiche in einem bestimmten Bereich daraufhin überprüfen, ob sie das tun, was sie sollen, oder anders ausgedrückt, ob der Speicher wirklich speichert. Aufgrund des Hardwareaufbaus äußern sich Speicherfehler meist derart, daß bestimmte Speicherstellen immer eine Null oder eine Eins an einer bestimmten Bitposition enthalten. Es ist daher zum Austesten nicht ausreichend, nur eine Null abzuspeichern und zu kontrollieren, ob sie anschließend noch vorhanden ist. Ein derartiger Test erkennt nicht den Fehler, daß ein Bit dauernd auf Null steht. Einige Fehler äußern sich sogar nur bei einem bestimmten Bitmuster, so daß ein ausführlicher Test alle möglichen Bitkombinationen verwenden muß.

Der erste Teil des Programmes befaßt sich mit der Ausgabe von Informationen über das Terminal. Das Unterprogramm WRCH gibt das in D0 gespeicherte Zeichen aus.

```
A_CTRL   EQU      $840021        Kontrollport des ACIAs
A_DATA   EQU      $840023        Datenport des ACIAs

WRCH     BTST     #1,A_CTRL      Ist der ACIA bereit ?
         BEQ.S    WRCH           Sprung, falls nicht
         MOVE.B   D0,A_DATA      Ausgabe des Zeichens
         RTS
```

Nun können wir ein Unterprogramm WRITES schreiben, um einen ganzen Text auszugeben. Register A1 zeigt auf den Anfang des Textes, der durch den Wert Null begrenzt wird.

```
WRITES   MOVEM.L  D0/A1,-(SP)    Retten der Register
WRS1     MOVE.B   (A1)+,D0       Holen eines Zeichens
         BEQ.S    WRS2           Aussprung, falls Ende erreicht
         BSR.S    WRCH           Ausgabe des Zeichens
         BRA.S    WRS1           Nächstes Zeichen
WRS2     MOVEM.L  (SP)+,D0/A1    Register wiederherstellen
         RTS                     und Rücksprung
```

Wir wollen alle möglichen Bitmuster in alle in dem angegebenen Bereich liegenden Speicherstellen schreiben. Danach müssen wir erneut durch den Speicher gehen, um zu prüfen, ob sich dieser Wert nirgends geändert hat. Es reicht nicht aus, diesen Test unmittelbar, nachdem wir eine Speicherstelle beschrieben haben, durchzuführen, da das Beschreiben einer Speicherstelle unerwartet den Zustand einer anderen ändern kann. Die Schleife selbst ist bei diesen beiden Vorgängen identisch, nur die in ihr vorgenommenen Operationen unterscheiden sich.

Ein sehr hübscher Weg, dies durchzuführen, besteht darin, in einem eigenen Unterprogramm den Speicherbereich zu traversieren. Für jede Adresse ruft dieses dann ein anderes Unterprogramm auf, das die notwendigen Operationen durchführt. Das erste Unterprogramm wollen wir SCAN nennen. Register A2 enthält bei Aufruf die Adresse des Unterprogramms, das die eigentliche Operation (Schreiben oder Testen) vornimmt. D0 enthält das momentane Bitmuster, und A0 wird als Zeiger in den zu testenden Bereich verwendet. Dieser ist durch die Konstanten MEMLO und MEMHI begrenzt. Diese beiden Register werden auch von den beiden möglichen Unterprogrammen verwendet, A0 wird bei jedem Aufruf eines dieser Unterprogramme erhöht.

```
SCAN    MOVE.L   A0,-(SP)       A0 retten
        MOVEA.L  #MEMLO,A0      Anfang des zu testenden Bereichs
SCN1    JSR      (A2)           Aufruf des entsprechenden
*                               Unterprogramms

* A0 wird in diesem Unterprogramm erhöht

        CMPA.L   #MEMHI         Ende der Schleife erreicht ?
        BNE.S    SCN1           Nein → weiter
        MOVE.L   (SP)+,A0       A0 wiederherstellen
        RTS                     Rücksprung
```

Die beiden von SCAN aufgerufenen Unterprogramme sind recht einfach aufgebaut. Das erste legt nur den Wert in D0 in der durch A0 adressierten Speicherstelle ab und inkrementiert A0.

```
WRITE   MOVE.B   D0,(A0)+       Den Wert speichern
        RTS
```

Das zweite prüft, ob der in D0 vorhandene Wert mit dem in der durch A0 adressierten Speicherstelle übereinstimmt. A0 wird inkrementiert, und eine Meldung wird ausgegeben, falls ein Fehler entdeckt wird.

```
READ    CMP.B    (A0)+,D0       Speichertest
        BEQ.S    RD1            Falls in Ordnung, Rücksprung

* Speicherfehler → Ausgabe einer Meldung

        MOVE.L   A1,-(SP)       A1 retten
        LEA.L    MESS3,A1       Zeiger auf die Fehlermeldung
        BSR.S    WRITES         Ausgabe der Meldung
        MOVE.L   (SP)+,A1       A1 wiederherstellen
RD1     RTS                     und zurück
```

Der letzte Schritt besteht darin, ein Hauptprogramm zu unseren Unterprogrammen zu schreiben. Wir gehen davon aus, daß der ACIA bereits vom Monitor oder vom Betriebssystem initialisiert wurde. Während der Programmausführung wird das aktuelle Testmuster in D0 abgelegt. Dieses Register wird mit $FF vorbelegt; der Befehl DBcc steuert die Schleife, die alle Bitmuster durchläuft. Wir benutzen den MOVE-Befehl im Wortformat zur Initialisierung des Bitmusters, da DBcc die unteren 16 Bits des Regis-

ters dekrementiert. Weiterhin zeigt A1 auf eine Meldung, die bei jedem Schleifendurchlauf ausgegeben wird, um zu zeigen, daß das Programm noch läuft.

```
MEMLO    EQU      $4000          Anfangsadresse des Bereiches
MEMHI    EQU      $8000          Endadresse
CR       EQU      $0D            ASCII-Zeichen    Return
LF       EQU      $0A            ASCII-Zeichen    Zeilenvorschub

MCHECK   LEA.L    MESS1,A1       Anfangsmeldung
         BSR.S    WRITES         Ausgeben
         LEA.L    MESS2,A1       Meldung bei Schleifendurchgang
         MOVE.W   #$FF,D0        Anfangs-Bitmuster

* Schleifenanfang

LOOP     BSR.S    WRITES         "Schleifendurchgang beendet"
*                                ausgeben
         LEA.L    WRITE,A2       Zeiger auf Unterprogramm WRITE
         BSR.S    SCAN           Durchlauf zum Schreiben
         LEA.L    READ,A2        Zeiger auf Unterprogramm READ
         BSR.S    SCAN           Durchlauf zum Testen
         DBRA     D0,LOOP        Dekrementieren des Bitmusters

* Test beendet. Ausgabe einer Schlußmeldung

         LEA.L    MESS4,A1       Zeiger auf die Nachricht
         BSR.S    WRITES         Ausgeben
         RTS                     Rücksprung in den Monitor

* Meldungen

MESS1    DC.B     'Start des Speichertests',CR,LF,0
MESS2    DC.B     'Schleifendurchgang beendet',CR,LF,0
MESS3    DC.B     'FEHLER gefunden',CR,LF,0
MESS4    DC.B     'Ende des Speichertests',CR,LF,0

         END
```

5
Arithmetik

Bisher haben wir bereits sehr viele Befehle des 68000 kennengelernt, doch Befehle zur Arithmetik waren bisher keine darunter. Dies ist beileibe kein Zufall - die meisten Computer verbringen mehr Zeit mit dem Hin- und Herschieben von Daten als mit dem Addieren irgendwelcher Zahlen. Der Gedanke, ein Computer sei eine bessere Rechenmaschine, ist reichlich altmodisch.

5.1 ADDITION

Zwar haben wir bereits Möglichkeiten kennengelernt, um einfache Additionen durchzuführen, da der 68000 bei der Auswertung von Adressen auch Additionen vornimmt. Solange mindestens einer der Summanden in einem Adreßregister vorliegt, können wir mit LEA und PEA Additionen durchführen, wie wir gesehen haben. Allerdings ist dies nur ein ab und zu nützlicher Trick; die meisten arithmetischen Befehle operieren auf Datenregistern.

Der Additionsbefehl heißt ADD, was wohl niemanden überraschen wird. Wie bei vielen anderen Befehlen des 68000 gibt es eine ganze Familie von ADD-Befehlen.

Der einfache ADD-Befehl wird mit mindestens einem Datenregister verwendet. Ist das Datenregister der Zieloperand, so kann jede Adressierungsart benutzt werden. Ist das Datenregister der Quelloperand, so muß der Zieloperand 'memory-alterable' adressiert werden. Der Befehl kann mit allen Längen verwendet werden, solange der Quelloperand kein Adreßregister ist. In diesem Fall ist die Verwendung des Byteformats unzulässig.

Entsprechend dem Ergebnis der Operation werden die Condition-Codes gesetzt. Das N- bzw. das Z-Bit werden gesetzt, falls das Ergebnis negativ bzw. Null war. Das V-Bit wird gesetzt, falls bei der Operation ein Überlauf auftrat, und das C- und das X-Bit werden beide gesetzt, wenn ein Übertrag entstand. Alle nicht explizit gesetzten Bits werden gelöscht.

Der Befehl ADD kann benutzt werden, um nur die unteren 16 oder 8 Bits des Ziel-Datenregisters zu modifizieren. Sein naher Verwandter ADDA wird zum Addieren von Werten in Adreßregistern verwendet, er verändert wie MOVEA keinen der Condition-Codes und kann nicht im Byteformat verwendet werden. Die Verwendung des Wortformats betrifft nur die Länge des Quelloperanden, der auf 32 Bit sign-extended und zu den 32 Bits im Zielregister addiert wird. Dabei muß unbedingt beachtet werden, daß bei Addition eines Wertes im Wortformat mit gesetztem oberen Bit ein negativer Wert zum Zieloperanden addiert wird.

Der Quelloperand für ADDA kann in jeder beliebigen Adressierungsart angegeben geben, aber auch hier gibt es einen Extrabefehl zur Addition von Direktoperanden. Der Befehl ADDI hat einen Direktoperanden als Quelloperanden, sein Zieloperand kann in jeder Adressierungsart vom Typ 'data-alterable' angegeben werden. Daher kann mit ADDI wohl ein Direktoperand zu einem Datenregister, nicht aber zu einem Adreßregister addiert werden. In diesem Fall kann jedoch alternativ der ADD-Befehl verwendet werden.

ADDI kann in jedem Größenformat benutzt werden und setzt die Condition-Codes ebenso wie ADD. Die am häufigsten verwendete Konstante zur Addition ist Eins, wenn z.B. eine Speicherstelle oder ein Register bei jedem Schleifendurchlauf inkrementiert wird. Einige Befehlssätze enthalten einen speziellen 'Inkrement'-Befehl, der 68000 aber geht viel weiter. Der Befehl ADDQ (add quick) kann verwendet werden, um einen beliebigen, 'alterable' adressierten Wert um eins bis acht zu erhöhen. Dies ist wesentlich praktischer als ein einfaches Inkrementieren, da z.B. ein als Zeiger dienendes Register oft um vier erhöht werden muß.

Bis auf zwei Unterschiede verhält sich ADDQ wie ADDI. Zunächst einmal ist er kürzer, und sollte daher ADDI vorgezogen werden, vor allem wenn das Vollwortformat verwendet wird. Die Condition-Codes werden in jedem Fall genauso wie bei ADDI gesetzt. Der zweite Unterschied besteht darin, daß als Zieloperand ein Adreßregister angegeben werden kann, wobei sich ADDQ wie ADDA mit der Angabe eines Direktoperanden verhält. Hier sind dann nur Wort- und Vollwortformat erlaubt, und die Condition-Codes werden nicht beeinflußt. Das ganze Adreßregister wird verändert, egal, welche Größe bei dem Befehl spezifiziert wird.

Als letztes Mitglied dieser Familie ist ADDX (add extended) zu nennen. Dieser Befehl hat zwei unterschiedliche Erscheinungsformen, abhängig davon, ob die Operanden Datenregister oder indirekt im Predecrementmodus adressierte Speicherstellen sind. In beiden Befehlen sind sämtliche Längenangaben erlaubt.

Der Befehl ADDX addiert bei Ausführung die Operanden genauso wie ADD, berücksichtigt dabei aber das X-Bit im Statusregister. Dieses kann von einer anderen arithmetischen Operation vor Aufruf des ADDX-Befehls gesetzt oder gelöscht werden und erlaubt so die Ausführung arithmetischer Operationen mit mehrfacher Genauig-

keit. Denken Sie daran, daß das X-Bit bei Rechenoperationen auf den gleichen Wert wie das C-Bit gesetzt wird, aber nicht wie dieses von anderen Befehlen wie z.B. MOVE beeinflußt wird.

Die Condition-Codes werden bei ADDX mit einer Ausnahme genauso wie bei ADD gesetzt. Diese Ausnahme betrifft das Z-Bit, daß zwar gelöscht wird, wenn das Ergebnis von Null verschieden ist, aber bei einem Ergebnis Null unverändert bleibt, anstatt wie sonst üblich gesetzt zu werden. Dieser Effekt ist zur Verwendung bei Operationen mit mehrfacher Genauigkeit gedacht. Wenn eines der Zwischenergebnisse von Null verschieden war, so wird das Z-Bit gelöscht, und es ist auch noch am Ende der kompletten Operation gelöscht. Ist das Z-Bit am Ende einer Operation mit mehrfacher Genauigkeit hingegen gesetzt, so ist das komplette Ergebnis dieser Operation Null.

5.2 SUBSTRAKTION

Die Befehlsfamilie SUB hat die gleichen Mitglieder wie die obenbeschriebene ADD-Familie. Der einfache SUB-Befehl hat ein Datenregister als Quell- oder Zieloperanden und setzt die Condition-Codes; das C- und das X-Bit werden diesmal bei Auftreten eines Borrows gesetzt.

SUBA wird mit einem Adreßregister als Zieloperanden angewendet, die Condition-Codes werden bei seiner Ausführung nicht verändert. SUBI wird bei Verwendung eines Direktoperanden benutzt, und SUBQ kann benutzt werden, wenn dieser Direktoperand im Bereich von eins bis acht liegt. Auch hier sollten Sie wieder daran denken, daß Sign-Extension durchgeführt wird, wenn der Befehl im Wortformat mit einem Adreßregister als Zieloperanden angegeben wird.

Selbstverständlich steht auch der Befehl SUBX zur Verfügung. Dieser subtrahiert zunächst die Quelle vom Ziel, anschließend wird von diesem Resultat noch das X-Bit abgezogen. Das Gesamtergebnis wird im Zieloperanden abgelegt. Die Operanden können auch hier nur ein Paar von Datenregistern oder ein Paar indiziert im Predecrementmodus adressierte Speicherstellen sein. Das Z-Bit wird nur gelöscht, wenn das Ergebnis von Null verschieden ist, ansonsten wird es nicht beeinflußt.

Wir haben zwar der Vollständigkeit halber alle Formen der Befehle aufgeführt, aber viele Assembler wählen automatisch die richtige Form aus, soweit ihnen dies möglich ist. Der Unterschied zwischen den Befehlen ist daher eher dann wichtig, wenn Code ohne Hilfe eines Assemblers erzeugt wird, wie dies etwa bei einem Compiler der Fall ist.

5.3 NEGIEREN

Mit Hilfe des Befehls NEG können beliebige Werte negiert werden. Dieser subtrahiert einfach den Quelloperanden von Null. Der Befehl kennt alle Größenformate und erlaubt für den Zieloperanden alle 'data-alterable' adressierbaren Werte.

Ist das Resultat dieser Operation Null, so wird das Z-Bit des Statusregisters gesetzt; C- und X-Bit werden gelöscht. Ist das Ergebnis von Null verschieden, so wird das Z-Bit zurückgesetzt, und C- und X-Bit werden gesetzt. N- und V-Bit werden wie üblich gesetzt bzw. gelöscht.

Der NEG-Befehl hat nur eine Variante, nämlich NEGX. Dieser negiert den angegebenen Wert und subtrahiert dann noch das X-Bit. Die Condition-Codes N und V werden genauso wie bei NEG gesetzt oder gelöscht. Z wird gelöscht, wenn das Ergebnis ungleich Null war, ansonsten bleibt es unberührt, ebenso wie bei ADDX und SUBX. C- und X-Bit werden abhängig von Auftreten eines Borrows gesetzt, sonst gelöscht. Der NEGX-Befehl wird normalerweise angewendet, um einen Wert mehrfacher Genauigkeit zu negieren, da dieser mehr als ein Vollwort belegt.

5.4 MULTIPLIKATION

Die Additions- und Subtraktionsbefehle des 68000 sind in dem Sinne vollständig, daß sie in allen drei Längenformaten anwendbar sind. Leider gilt das nicht für die Multiplikation und die Division, da bei diesen Befehlen nur das Wortformat erlaubt ist. Vorläufige Informationen von Motorola lassen jedoch darauf schließen, daß das Vollwortformat dieser Befehle beim 68020 verfügbar sein wird.

Zur Multiplikation stehen zwei Befehle, MULS und MULU, zur Verfügung. Der einzigen Unterschied zwischen diesen beiden besteht darin, daß MULS die Vorzeichen der Operanden beachtet und auch ein signiertes Resultat liefert, wogegen MULU ohne Beachtung der Vorzeichen rechnet.

Beide Befehle verwenden einen beliebigen Operanden vom Typ 'data' als Quell- und ein Datenregister als Zieloperanden. Die unteren 16 Bits des Zielregisters werden mit dem als Wort adressierten Quelloperanden multipliziert. Ist dies eine Stelle im Speicher, so werden hierfür wie bekannt die an dieser Stelle liegenden 16 Bits verwendet, ist es ein Datenregister, so werden die unteren 16 Bits benutzt.

Das Ergebnis wird als 32-Bit-Wert im Zielregister abgelegt. N- und Z-Bit werden wie üblich gesetzt, wenn das Resultat Null oder negativ ist. Bei MULU bedeutet negativ, daß das oberste Bit gesetzt ist. V- und C-Bit werden auf jeden Fall gelöscht, das X-Bit bleibt wie es ist.

5.5 AUSTAUSCH VON REGISTERN

An dieser Stelle ist ein weiterer nützlicher Befehl aufzuführen. SWAP vertauscht in einem Datenregister die oberen 16 Bits mit den unteren. Dieser Befehl ist sehr sinnvoll bei Multiplikations- und Divisionsroutinen einzusetzen, wo man sich mit den vorhandenen 16-Bit-Operationen herumplagen muß.

SWAP setzt bei der Ausführung die Condition-Codes wie folgt: Das N-Bit wird gesetzt, wenn das höchste Bit in dem Register nach dem Austausch gesetzt ist, ansonsten wird es gelöscht. Das getestete Bit war vorher das höchstwertige Bit des unteren Worts dieses Registers. Das Z-Bit wird gesetzt bzw. gelöscht, wenn der Inhalt des Registers Null bzw. davon verschieden ist. V- und C-Bit werden immer gelöscht, das X-Bit bleibt unberührt.

Zwei andere Befehle sollten auch an dieser Stelle eingeführt werden. Der erste heißt EXG und tauscht bei Ausführung einfach die Inhalte der beiden als Operanden spezifizierten Register aus. Diese Register können Daten- wie Adreßregister sein, auch die gemischte Verwendung ist hier erlaubt. Da das ganze Register ausgetauscht wird, gibt es diesen Befehl nur im Vollwortformat. Die Bits des Statusregisters werden von diesem Austausch nicht beeinflußt.

Der andere Befehl heißt EXT und führt mit dem Inhalt eines Datenregisters Sign-Extension durch. Diesen Befehl gibt es nur im Wort- und Vollwortformat; bei Verwendung des Wortformats wird das höchste Bit des unteren Bytes (also Bit 7) in die Bits 8 - 15 kopiert; beim Vollwortformat wird das höchste Bit des unteren Worts (Bit 15) in die Bits 16 - 31 kopiert.

Der EXT-Befehl wird beispielsweise verwendet, nachdem ein eine signierte Zahl darstellendes Byte in ein Register geladen wurde, um zu einem Wort- oder Vollwortoperanden addiert zu werden. Bei Addition mit einem Wortoperanden muß dieses Byte zunächts mit EXT.W auf Wortformat gebracht werden, bei Addition mit einem Vollwortoperanden mittels EXT.L auf Vollwortformat. Bei vielen Befehlen wird die Sign-Extension von Wort- auf Vollwortformat automatisch durchgeführt, vor allem, wenn Adreßregister mit im Spiel sind.

Der EXT-Befehl beeinflußt das X-Bit nicht; das V- und das C-Bit werden gelöscht, das N- und das Z-Bit werden wie üblich in Abhängigkeit vom Ergebnis gesetzt.

5.6 MULTIPLIKATION VON VOLLWORTEN

Da leider kein Befehl zur Multiplikation von Vollworten vorhanden ist, müssen wir dafür ein Unterprogramm verwenden. Dieses multipliziert zwei 32-Bit-Faktoren zu einem 32-Bit-Produkt. Wenn wir natürlich zwei große 32-Bit-Werte multiplizieren,

wird bei der Multiplikation ein Überlauf auftreten. Diesen wollen wir in diesem ersten Beispiel zunächst unbeachtet lassen.

Um dieses Unterprogramm schreiben zu können, sollten wir uns zunächst daran erinnern, wie wir Multiplikationen von Hand ausführen. Wenn Sie nicht immer auf einen Taschenrechner ausweichen, werden Sie sich sicher an den Algorithmus erinnern, der zur Multiplikation zweier zweistelliger Dezimalzahlen benutzt wird. Einstellige Zahlen kann jeder noch im Kopf multiplizieren, aber bei allem, was darüber hinaus geht, wird normalerweise der Ruf nach Papier und Bleistift laut. Wir wollen im folgenden einfach mal die Zahlen AB und CD mittels der sogenannten 'kreuzweisen' Multiplikation multiplizieren. Jeder Buchstabe steht dabei für eine Ziffer zwischen 0 und 9.

```
          A      B
          C      D
        ---------------
         D*A    D*B
C*A      C*B
-------------------------
C*A  D*A+C*B  D*B
-------------------------
```

Da wir leichter einzelne Ziffern miteinander multiplizieren können, spalten wir die Multiplikation in eine Reihe einfacherer Multiplikationen und Additionen auf. Die höchste Ziffer des Produkts ergibt sich als Produkt der beiden höchsten Ziffern der Faktoren, und die niedrigste Ziffer ebenso als Produkt der beiden niedrigsten Ziffern. Die dazwischenliegende Ziffer ist die Summe der beiden 'Kreuzterme'. Natürlich müssen wir hierbei den Überlauf behandeln, wenn das Produkt zweier Ziffern zweistellig ist; dies wird durch Übertragen der überschüssigen Ziffer in die nächste Spalte erledigt.

Dieser Algorithmus ist exakt das, was wir für unsere Multiplikationsroutine benötigen. Der 68000 kann Multiplikationen auf Worten ohne weiteres ausführen, und so spalten wir die Multiplikation von Vollworten auf in eine Folge von Multiplikationen und Additionen von Worten. Im folgenden betrachten wir daher jeden 32-Bit-Wert als aus zwei »16-Bit-Ziffern« bestehend, die wir mit RH und RL bezeichnet werden. R ist hierbei durch den Namen des 32-Bit-Registers zu ersetzen. Dann können wir den oben beschriebenen Algorithmus anwenden, wobei wir zunächst einmal die durch C*A bezeichnete dritte Ziffer übersehen wollen. Liegen die beiden Faktoren in den Registern D1 und D2 vor, so ergibt sich das Produkt wie folgt:

RL = D1L×D2L
RH = (D2H×D1L) + (D1H×D2L) + Übertrag von RL

Dies wollen wir nun in 68000-Befehle umsetzen. Das zu schreibende Unterprogramm soll die beiden in D1 und D2 liegenden Vollworte multiplizieren und das Ergebnis in D1 zurückgeben.

```
MUL     MOVEM.L  D2-D4,-(SP)     Register retten
        MOVE.W   D1,D3           D1L nach D3
        MOVE.W   D2,D4           D2L nach D4
        SWAP     D1              D1H nach D1
        SWAP     D2              D2H nach D2

* Multiplizieren

        MULU     D3,D2           D2 = D1L * D2H
        MULU     D4,D1           D1 = D2L * D1H
        MULU     D4,D3           D3 = D2L * D1L

* Addieren der Terme ohne Rücksicht auf Überlauf

        ADD.W    D2,D1           D1 = D2L * D1H + D1L * D2H

* Kreuzsumme als obere Ziffer abspeichern

        SWAP     D1              D1 = Kreuzsumme
        CLR.W    D1              Untere Ziffer löschen

* Untere Ziffer eintragen

        ADD.L    D3,D1           D1L und D1H stimmen nun
        MOVM.L   (SP)+,D2-D4     Register wiederherstellen
        RTS
```

Zum Verständnis dieses Unterprogramms ist es wichtig, daran zu denken, daß MULU ein 32-Bit-Ergebnis liefert. In der ersten Zeile retten wir (wie gehabt) die Inhalte der Arbeitsregister; in den nächsten vier Zeilen teilen wir die beiden Faktoren in vier 16-Bit-Werte auf. Die oberen Worte dieser Register werden von MULU ignoriert, so daß ihr Inhalt belanglos ist. Der Befehl MOVE.W beeinflußt nur das untere Wort, so daß wir mit SWAP das obere mit dem unteren Wort vertauschen müssen.

Die nächsten 3 Zeilen ergeben drei 32-Bit-Werte als Ergebnisse der Multiplikation von 16-Bit-Werten. Die Kreuzprodukte addieren wir dann mittels ADD.W, wobei wir den evtl. auftretenden Überlauf ignorieren. Die 16-Bit-Summe wird dann in das obere Wort des Ergebnisregisters D1 kopiert, und dessen unteres Wort wird gelöscht.

Die letzte Aktion besteht darin, das untere Wort des Ergebnisses an die richtige Stelle im Resultatsregister zu bringen. Dies kann aber nicht einfach mit einem MOVE.W von D3 nach D1 erreicht werden, da ein eventueller Übertrag von der unteren Stelle beachtet werden muß. Das Produkt von D1L und D2L liegt in D3 vor; ein einfacher ADD.L von D3 und D1 garantiert, daß das untere Wort richtig im Ergebnis eingetragen wird. Jeglicher Überlauf hierbei wird wiederum ignoriert.

Die nun folgende Routine soll die beiden 32-Bit-Werte in Register D0 und D1 zu einem 64-Bit-Ergebnis im Registerpaar D6/D7 multiplizieren. Anders ausgedrückt, soll D6 nach der Ausführung des Unterprogramms das höherwertige und D7 das niederwertige Vollwort des Ergebnisses enthalten. Der obige Algorithmus muß dafür wie folgt erweitert werden:

D7L = D1L×D2L
D7H = (D2H×D1L) + (D1H×D2L) + Übertrag von D7L
D6L = D1H×D2H + Übertrag von D7H
D6H = Übertrag von D6L

Nunmehr verwenden wir auch das X-Bit, das angibt, ob bei der letzten Rechenoperation ein Übertrag auftrat.

```
LMUL    MOVEM.L  D3/D4,-(SP)    Register retten
        MOVE.L   D1,D          3          D1 nach D3
        MOVE.L   D2,D4            D2 nach D4
        SWAP     D3               D3 = D1H
        SWAP     D4               D4 = D2H

* Multiplizieren

        MOVE.W   D1,D7            D7 = D1L
        MULU     D2,D7            D7 = D2L * D1L
        MOVE.W   D3,D6            D6 = D1H
        MULU     D4,D6            D6 = D2H * D1H
        MULU     D1,D4            D4 = D1L * D2H
        MULU     D2,D3            D3 = D2L * D1H

* Addieren der Kreuzprodukte

        ADD.L    D3,D4            X-Bit ist nunmehr relevant

* Behandeln des unteren Worts des Kreuzprodukts

        MOVE.W   D4,D3            D3L = D4L
        SWAP     D3               D3H = D4L
        CLR.W    D3               D3L = 0

* Behandeln des oberen Worts des Kreuzprodukts

        CLR.W    D4               D4L = 0
        ADDX.W   D3,D4            D4L = Zustand des X-Bits, da
*                                 D3L Null ist
        SWAP     D4               D4 enthält die oberen 17 Bit des
*                                 Kreuzprodukts

* Addition der unteren Worte des Kreuzprodukts in D3H nach D7

        ADD.L    D3,D7            X-Bit relevant

* Addition der oberen Worte des Kreuzprodukts in D4

        ADDX.L   D4,D6            Berücksichtigung des Übertrags
*                                 vom vorherigen ADD

* D6, D7 enthalten nun das unsignierte 64-Bit Produkt
```

```
        TST.L   D1              War D1 negativ ?
        BPL.S   LMUL1           Sprung, falls nicht
        SUB.L   D2,D6           D2 vom Ergebnis abziehen
LMUL1   TST.L   D2              War D2 negativ ?
        BPL.S   LMUL2           Sprung, falls nicht
        SUB.L   D1,D6           D1 vom Ergebnis abziehen
LMUL2   MOVEM.L -(SP),D3/D4     Register wiederherstellen
        RTS
```

In den ersten paar Zeilen retten wir nur die Inhalte unserer Arbeitsregister und verteilen die höherwertigen Worte unserer Faktoren auf die unteren Worte der Arbeitsregister.

In den nächsten Zeilen werden die Produkte gebildet. Die unterste 'Ziffer' des Gesamtergebnisses liegt nun bereits in D7 vor. Analog dazu liegen in D6 bereits (bis auf die Überträge) die beiden höchstwertigen Ziffern vor, die Kreuzprodukte werden in den Arbeitsregistern D3 und D4 abgelegt.

Diese werden dann in D4 zu einem 32-Bit-Wert addiert, wobei das X-Bit einen Überlauf anzeigt. Das X-Bit müssen wir dann später noch zu D6 addieren. Die nächsten Zeilen behandeln dann das Kreuzprodukt. Dessen unteres Wort muß in D7H, der zweiten Stelle des Gesamtergebnisses, abgelegt werden. Außerdem muß der in D7H stehende Übertrag aus der Multiplikation der beiden niederwertigen Worte berücksichtigt werden. Daher kopieren wir D4L nach D3L, vertauschen die Registerhälften und löschen das untere Wort, so daß wir nun das untere Wort des Kreuzproduktes in D3H zur Verfügung haben. Dieses muß nun zu D7 addiert werden, jedoch muß vorher noch das X-Bit verarbeitet werden, das noch den Übertrag der Addition der Kreuzprodukte enthält. Das höherwertige Wort des Kreuzproduktes muß in die untere Hälfte eines Arbeitsregisters geladen werden, bevor es zu D6 addiert werden kann; dabei muß auch der Zustand des X-Bits beachtet werden. Dieses wird nun nach D4L geholt, das zuvor mittels CLR.W gelöscht wird. Das Übertragen des X-Bits kann nicht mit einem ADDX mit Direktoperand 0 realisiert werden, da ADDX nur mit zwei Daten- oder Adreßregistern als Operanden benutzt werden darf. Da aber D3L gerade eben erst gelöscht wurde, können wir ADDX.W mit den Operanden D3 und D4 aufrufen, was der Verwendung des Direktoperanden Null entspricht. Danach enthält D4H die höherwertige Ziffer des Kreuzprodukts, und D4L Eins oder Null, je nach dem Zustand des X-Bits. Ein einfacher SWAP bewirkt nun, daß D4 die oberen 17 Bit des Kreuzprodukts enthält, also auch den Übertrag.

Inzwischen haben wir unser Ziel beinahe erreicht. Durch Addition von D3 und D7 erhalten wir die korrekten unteren beiden Ziffern des Gesamtergebnisses, wobei das X-Bit einen eventuellen Übertrag von D7 nach D6 angibt. Mittels ADDX.L wird dann mit D4 das höhere Wort des Kreuzprodukts und der Übertrag zu dem bereits vorher generierten Produkt der oberen Worte der Faktoren in D7 addiert, wodurch nun das gewünschte Ergebnis in D6 und D7 vorliegt.

Dies funktioniert bei positiven Zahlen gut; aber bisher haben wir nur eine unsignierte Multiplikation mit den Eingabewerten durchgeführt. Hätten wir beispielsweise −1 mit 2 multipliziert, so hätten wir tatsächlich $FFFFFFFF mit $2 multipliziert. Bei der verwendeten unsignierten Arithmetik ergibt dies $1 in D6 und $FFFFFFFE in D7, und das entspricht in keinster Weise dem Wert −2 der signierten Arithmetik ($FFFFFFFF in D6 und $FFFFFFFE in D7).

Die notwendige Korrektur ist recht einfach vorzunehmen; sie basiert auf der Berechnung im Zweierkomplement. −A wird hier durch M − A dargestellt, wobei M für 2 hoch 32 (= $100000000) steht. Daher gilt:

$$(-A)\times B = (M-A)\times B = (M\times B)-(A\times B)$$

Das unsignierte Produkt einer negativen und einer positiven Zahl ist daher gleich dem signierten Produkt plus M*B, was der um 32 Stellen verschobenen 64-Bit-Darstellung von B entspricht. Da B in einem 32-Bit-Register und das bisherige Ergebnis im Registerpaar D6/D7 vorliegt, subtrahieren wir einfach B vom höherwertigen Register D6. Da die ursprünglichen Operanden nach wie vor in den Registern D1 und D2 vorhanden sind, müssen wir D2 von D6 subtrahieren, wenn D1 negativ ist. Analog müssen wir D1 von D6 abziehen, wenn D2 negativ ist. Diese Korrektur vollendet dann unsere signierte Multiplikationsroutine.

5.7 DIVISION

Wir haben bereits erwähnt, daß der 68000 keine Befehle zur Division von Langworten besitzt, sondern nur zwei Befehle für signierte bzw. unsignierte Division. Der Quelloperand ist in beiden Fällen ein Wort, das in jeder Adressierungsart vom Typ 'data' angegeben werden kann, und der Zieloperand ist stets ein komplettes Datenregister. Der im Datenregister vorliegende 32-Bit-Wert wird komplett durch den als Quelloperanden spezifizierten Wert dividiert. Wie erwartet, ist DIVU für die unsignierte und DIVS für die signierte Division zuständig.

In beiden Fällen werden zwei Ergebnisse erzeugt. Das untere Wort des Zielregisters enthält nach der Ausführung den Quotienten, wenn dieser in ein Wort paßt. Die obere Hälfte des Registers enthält den Divisionsrest. Bei Verwendung von DIVS hat dieser Rest das gleiche Vorzeichen wie der Dividend.

Das N- und das Z-Bit werden wie üblich gesetzt oder gelöscht, das C-Bit wird immer gesetzt, das X-Bit behält seinen alten Wert. Ist der Quotient zu groß, um in ein Wort zu passen, tritt ein Überlauf auf und das V-Bit wird gesetzt. Da der Überlauf vom 68000 während der Abarbeitung des Befehls erkannt wird, sind in diesem Fall sowohl das Ergebnis als auch der Zustand von Z- und N-Bit nicht definiert und dürfen auf keinen Fall weiterverarbeitet werden. Der in Kapitel 7 beschriebene Befehl TRAPV kann

benutzt werden, um einen Trap zu verursachen, sobald ein Überlauf auftritt. Hatte der Quelloperand den Wert Null, so tritt ein 'Division durch Null'-Trap auf.

5.8 DIVISION VON VOLLWORTEN

Da DIVS und DIVU nur dann anwendbar sind, wenn das Ergebnis in ein Wort paßt, benötigen wir nun auch noch ein Unterprogramm zur Division von Vollworten. Dieses ist etwas komplizierter als die oben beschriebene Multiplikationsroutine. Das folgende Unterprogramm stammt von Dr. Arthur Norman; es dividiert den in D1 vorliegenden 32-Bit-Dividenden durch den 32-Bit-Divisor in D2. Der 32-Bit-Quotient wird in D1 zurückgegeben, D2 enthält den Divisionsrest.

Im ersten Abschnitt werden die Vorzeichen behandelt, so daß nachher die Hauptarbeit mit unsignierter Arithmetik durchgeführt werden kann. Zunächst überprüfen wir, ob der Divisor negativ ist. Falls dies der Fall ist, ändern wir einfach sein Vorzeichen, führen die Division durch und ändern anschließend das Vorzeichen des Ergebnisses.

```
DIV     TST.L   D2          Divisor < 0 ?
        BPL.S   DIV00       Nein → Sprung zur Division
        NEG.L   D2          D2 = - D2
        BSR.S   DIV00       Division
        NEG.L   D1          Resultat negieren
        RTS                 und zurück
```

Im nächsten Fall müssen wir uns nun um einen eventuell negativen Dividenden kümmern. Hierbei müssen wir das Vorzeichen des Dividenden ändern, die Division durchführen und anschließend sowohl das Vorzeichen des Quotienten als auch das des Restes ändern.

```
DIV00   TST.L   D1          Dividend > 0 ?
        BPL.S   DIV0        Ja → Sprung zur Division
        NEG.L   D1          Dividend = - Dividend
        BSR.S   DIV0        Division
        NEG.L   D1          Quotient = - Quotient
        NEG.L   D2          Rest = - Rest
        RTS                 und zurück
```

Im folgenden müssen wir uns nur noch mit vorzeichenlosen Zahlen beschäftigen. D1 und D2 sind größer oder gleich Null und kleiner als $80000000. Ist D2 tatsächlich Null, so tritt ein 'Division durch Null'-Fehler auf. Um uns (bzw. unserem Programm) unnötige Arbeit zu ersparen, prüfen wir zunächst eine Handvoll trivialer Fälle ab. Der erste dieser Fälle betrifft einen Divisor mit höchstens 16 signifikanten Bits. In diesem Fall können wir einfach den normalen DIVU-Befehl zur Division benutzen. Dies wird dann in dem Unterprogramm DIVX, welches auch die Register für Quotient und Rest richtig setzt, erledigt.

```
DIV0    CMPI.L   #$FFFF,D2      D2H = 0 ?
        BLS.L    DIVX           D2 <= 16 Bit → Unterprogramm
```

Zwei weitere Trivialfälle behandeln wir getrennt: Ist der Dividend größer als der Divisor, so lautet das Ergebnis Null; sind beide gleich, so lautet es Eins.

```
        CMP.L    D1,D2          Ist D2 <= D1 ?
        BEQ.S    DIV01          D1 = D2
        BLS.S    DIV02          D1 > D2 → komplette Rechnung

* Hier gilt nun D1 < D2; der Quotient ist Null

        MOVE.L   D1,D2          korrekter Rest
        MOVEQ    #0,D1          Quotient = 0

* D1 = D2; Quotient ist Eins

DIV01   MOVEQ    #0,D2          Rest = 0
        MOVEQ    #1,D1          Quotient = 1
        RTS
```

Im allgemeinen Fall besitzt der Divisor mehr als 16 signifikante Bits. Da der Dividend in einem 32-Bit-Register steht, besitzt der Quotient maximal 16 Bits. Wir generieren zunächst ein Näherungsergebnis, indem wir Dividend und Divisor so durch einen Skalierungsfaktor teilen, daß der skalierte Divisor nur noch 16 Bit lang ist. Mit diesen derart skalierten Operanden können wir dann eine ganz normale Division mittels DIVU durchführen. Als Skalierungsfaktor wählen wir 1 + (D2 / $10000). Dieser Faktor benötigt maximal 16 Bits, da D2 kleiner als $80000000 ist und unser Skalierungsfaktor somit als höchsten Wert $8001 annehmen kann.

```
DIV02   MOVEM.L  D3-D5,-(S)     Arbeitsregister retten
        MOVE.L   D2,D3          D3 = Divisor
        CLR.W    D3             D3L = 0
        SWAP     D3             D3 = D2 / $100000
        ADDQ.L   #1,D3          D3 = 1 + (D2 / $100000)

* Skalierungsfaktor in D3. Skalieren von Dividend und Divisor

        MOVE.L   D1,D4          D4 = Dividend
        MOVE.L   D2,D5          D5 = Divisor
        MOVE.L   D3,D2          D2 = Skalierungsfaktor
        BSR.S    DIVX           D1 = D1 / Skalierungsfaktor
        MOVE.L   D5,D2          Divisor wieder zurück
        DIVU     D3,D2          D2L = D2 / Skalierungsfaktor

* D2 besitzt nun maximal noch 16 Bits

        DIVU     D2,D1          Division der skalierten Terme
```

An dieser Stelle enthält D1L nun einen Schätzwert des gesuchten Quotienten. Dieses bisherige Ergebnis überprüfen wir nun, indem wir es mit dem ursprünglichen Divisor multiplizieren und das Produkt mit dem ursprünglichen Dividenden vergleichen. Im gleichen Schritt bestimmen wir auch den Divisionsrest. Stimmt unser Quotient

nicht, so addieren bzw. subtrahieren wir eine Eins von ihm und überprüfen das Ergebnis erneut. Dies führen wir solange durch, bis daß unser Quotient stimmt, d.h. der Rest kleiner als der Divisor ist.

```
        ANDI.L   #$FFFF,D1      D1H = 0
DIV03   MOVE.L   D5,D2          Divisor wieder zurück
        MOVE.L   D5,D3          Divisor auch nach D3
        SWAP     D3             D3L = D2H
        MULU     D1,D2          D2 = D1 * D2L
        MULU     D1,D3          D3 = D1 * D2H; D3H ist Null
        SWAP     D3             D3H = D1 * D2H; D3L ist Null
        ADD.L    D3,D2          Gesamtprodukt; Übertrag kann bei
*                               dieser Addition nicht auftreten
        SUB.L    D4,D2          Ursprünglichen Dividenden
*                               abziehen
        BHI.S    DIV04          Rest negativ: Zuviel des Guten
        NEG.L    D2             Vorzeichen wechseln
        CMP.L    D2,D5          Mit dem ursprünglichen Divisor
*                               vergleichen
        BHI.S    DIV05          Rest kleiner als Divisor
        ADDQ.L   #1,D1          Quotient inkrementieren
        BRA.S    DIV03          Erneuter Versuch
DIV04   SUBQ.L   #1,D1          Quotient dekrementieren
        BRA.S    DIV03          Erneuter Versuch

* Ergebnis stimmt

DIV05   MOVEM.L  (SP)+,D3-D5    Register wiederherstellen
        RTS
```

Nunmehr müssen wir nur noch das Unterprogramm DIVX schreiben. Dieses wird aufgerufen, wenn der Quotient garantiert in 16 Bit paßt (Sie erinnern sich, Fall 1) sowie zur Skalierung von Dividend und Divisor. Dieses Unterprogramm berechnet den Quotienten aus D1 und D2 und legt diesen in D1 ab; der Rest findet sich nachher in D2 wieder.

```
DIVX    MOVEM.W  D1/D3,-(SP)     D1L und D3L retten
        CLR.W    D1             D1L = 0
        SWAP     D1             D1L = D1H; D1H = 0
        DIVU     D2,D1          D1L = D1H / D2
        MOVE.W   D1,D3          Zwischenergebnis retten
        MOVE.W   (SP)+,D1L      D1L wiederherstellen

* D1H enthält D1H mod D2; D1L ist wie beim Einsprung

        DIVU     D2,D1          D1L = (D1L + ( D1H mod D2 ))/ D2
        SWAP     D1             D1L enthält nun den Rest
        MOVEQ    #0,D2          D2 = 0
        MOVE.W   D1,D2          Rest nach D2
        MOVE.W   D3,D1          D1L = oberes Wort des Quotienten
        SWAP     D1             Um den 32-Bit-Quotienten zu
*                               erhalten
        MOVE.W   (SP)+,D3       D3L wiederherstellen
        RTS                     Ende der Berechnung
```

Wie Sie sicher bemerkt haben, haben wir hier zwar mittels MOVEM.W die unteren Worte von D1 und D3 gerettet, diese jedoch nicht mit einem erneuten MOVEM wiederhergestellt. Ein Grund dafür ist, daß es bequemer ist, die Register an verschiedenen Stellen wieder vom Stack zu holen; die Hauptursache ist jedoch darin zu sehen, daß MOVEM.W nicht nur die unteren Worte, sondern durch Sign-Extension alle 32 Bits der Register verändert.

5.9 DEZIMALZAHLEN

Alles bisher Gesagte betraf Arithmetik mit binären Werten, d.h. Zahlen in Zweierkomplementdarstellung. Mit diesen Zahlen läßt's sich im Computer leicht und vor allem schnell rechnen. Die Konvertierung zwischen ihnen und für den Menschen lesbaren Dezimalzahlen ist jedoch ein aufwendige Sache.

Manche höhere Programmiersprache wie z.B. COBOL läßt dem Programmierer die Wahl des zu verwendenden Zahlensystem, binär oder dezimal. Der Vorteil der binären Arithmetik liegt ganz klar bei der Geschwindigkeit, aber der Prozeß der Umwandlung in und aus Dezimalzahlen ist recht langsam. Der Vorteil der dezimalen Arithmetik ist darin zu sehen, daß dezimale Eingaben ohne Konvertierung direkt im Computer weiterverarbeitet werden können. Diese Verarbeitung ist dann allerdings relativ langsam.

Der 68000 stellt 3 Befehle für Dezimalarithmetik zur Verfügung. Dies sind ABCD (add binary coded decimal), SBCD (subtract binary coded decimal) und NBCD (negate binary coded decimal). Sie arbeiten alle mit Byteoperanden, wobei ein Byte hier zwei Dezimalstellen im BCD-Code (binary coded decimal; binär codierte Dezimalzahlen) repräsentiert. Jedes Nibble (4 Bit) wird zur Darstellung einer Dezimalzahl zwischen 0 und 9 herangezogen. Die Dezimalzahl 15 hat daher die binäre Darstellung $0F, aber die Form $15 im BCD-Code.

Normalerweise wird eine Zahl von einer externen Einheit (z.B. einem Terminal) Zeichen für Zeichen in dezimaler Schreibweise eingegeben. Um diese Dezimalzahl in die binäre Darstellung umzuwandeln, benötigen wir eine Routine wie die folgende:

```
RDN     MOVEQ   #0,D1      D1 = 0
        MOVEQ   D1,D0
RDN1    BSR     RDCH      Ein Zeichen nach D0 einlesen
        SUB.B   #'0',D0   Das Zeichen '0' von D0 abziehen
        BMI.S   RDN2      Negativ, da das eingelesene Zeichen
*                         keine Ziffer war → Ende der Eingabe
        CMP.B   #9,D0     Zeichen größer als eine Ziffer ?
        BGT.S   RDN2      Ja → Ende der Eingabe
        MULS    #10,D1    D1 = 10 * D1
        ADD.L   D0,D1     D1 = 10 * D1 + D0
        BRA.S   RDN1      Nächste Ziffer lesen
RDN2    RTS               D1 enthält die eingelesene Zahl
```

Dieses Unterprogramm erstellt die binäre Form einer von einer externen Quelle eingelesenen Dezimalzahl. Das Unterprogramm RDCH dient dazu, ein einzelnes Zeichen von dieser Quelle einzulesen. Das Zeichen wird zunächst daraufhin getestet, ob es eine gültige Dezimalziffer darstellt, und dann in seine binäre Darstellung umgewandelt, indem vom Code des eingelesenen Zeichens der Code für das Zeichen '0' subtrahiert wird. (In allen üblichen Codes liegen die Ziffern von 0 bis 9 unmittelbar hintereinander.) Der bisher aufsummierte Wert wird dann mit 10 multipliziert und der binäre Wert der neuen Ziffer aufaddiert. Dieses Unterprogramm funktioniert nur dann korrekt, wenn die binäre Darstellung der einzulesenden Zahl in ein Wort paßt. Zum Einlesen größerer Zahlen wäre ein Aufruf unserer Multiplikationsroutine nötig. Ein ebenso komplexer Algorithmus wird benötigt,um die Zahl zwecks Ausgabe wieder in ihre dezimale Darstellung umzuwandeln.

All das benötigt Zeit, und es kann durchaus sein, daß die ganze Um- und Rückwandlung nur den Zweck hat, die so erhaltenen Zahlen zu irgendwelchen anderen Werten zu addieren und das Resultat wieder auszugeben. Beispielsweise kann es nötig sein, einfach nur eine Liste von (Dezimal-)Zahlen aufzuaddieren. Zu diesem Zweck könnten wir ein Unterprogramm schreiben, das unsere Routine zum Einlesen und Umwandeln der Dezimalzahlen benutzt.

```
COUNT   MOVEQ    #0,D4        D4 = 0
        MOVE.W   #4,D5        5 (4+1) Zahlen aufaddieren
CNT1    BSR.S    RDN          Zahl einlesen, Binärform nach D1
        ADD.L    D1,D4        Aufaddieren
        DBRA     D5,CNT1      Nächste Zahl oder Schluß
```

In diesem Fall wäre es jedoch wesentlich einfacher, dezimale Arithmetik zu benutzen. Zur Darstellung von Dezimalzahlen werden wir nunmehr Speicherbereiche verwenden. Das folgende Unterprogramm liest eine Dezimalzahl zeichenweise ein und legt sie im BCD-Format in dem durch A1 adressierten 8 Byte großen Speicherbereich ab. Hierbei nehmen wir an, daß die einzulesenden Zahlen rechtsbündig in einem Feld von 16 Zeichen dargestellt sind, so daß die Zahl genau in unseren BCD-Bereich paßt. Führende Leerzeichen werden als Null behandelt. Wir nehmen keinerlei Test vor, ob die eingelesenen Zeichen tatsächlich Ziffern oder Leerzeichen sind, sondern wandeln mit einem kleinen Trick alle Zeichen um. Sind unter den Zeichen auch andere als Ziffern, so stellt das Ergebnis nicht unbedingt die BCD-Codierung einer Dezimalzahl dar. A1 zeigt nach Abarbeitung des Unterprogramms auf das erste Byte hinter der BCD-Codierung.

```
DRDN    MOVEM.L  D0/D1,-(SP)    Register retten
        MOVEQ    #7,D1          Zähler initialisieren
DRDN1   BSR      RDCH           Zeichen nach D0 einlesen

* Unteres Nibble ins obere Nibble verschieben. Das obere Nibble
* steht dann im zweiten Byte des Registers
```

```
        ASL.B    #4,D0          Viermal links schieben
        MOVE.B   D0,(A1)        Im Speicher ablegen
        BSR      RDCH           Nächstes Zeichen holen
        AND.B    #$0F,D0        Oberes Nibble löschen
        OR.B     D0,(A1)+       Im Speicher ablegen
        DBRA     D0,DRDN1       Nächste Ziffer oder Ende
        MOVEM.L  (SP)+,D0/D1    Register wiederherstellen
        RTS
```

In den ersten beiden Zeilen retten wir die Register und initialisieren einen Zähler. Durch Aufruf von RDCH erhalten wir dann die Zeichendarstellung einer Ziffer oder des Leerzeichens in D0. Da im normalerweise verwendeten ASCII-Code die Ziffern durch die hexadezimalen Zahlen $30 bis $39 und das Leerzeichen durch $20 dargestellt werden, erhalten wir durch einfaches Ignorieren des oberen Nibbles den Wert der Ziffer, solange nur Zahlen oder Leerzeichen in der Eingabe auftreten (wovon wir hier ausgegangen waren). Dann schieben wir das Byte einfach um 4 Positionen nach links und erhalten so im oberen Nibble die eingelesene dezimale Ziffer (die rechts freiwerdenden Bitpositionen werden mit Nullen aufgefüllt). Dieses Byte speichern wir dann an der durch A1 adressierten Speicherstelle ab.

Ein erneuter Aufruf von RDCH liefert uns dann die nächste Ziffer bzw. das nächste Leerzeichen. Diesmal löschen wir das obere Nibble, indem wir das Byte mit dem Wert $F (binär 1111) maskieren ('und-verknüpfen' bzw. 'ANDen' im Sprachgebrauch mancher Programmierer). Dieses Nibble fügen wir als unteres dem bereits abgespeicherten oberen Nibble zu und erhalten so in der durch A1 adressierten Speicherstelle zwei Ziffern der eingelesenen Zahl in BCD-Codierung. Da wir hierbei Postincrementadressierung benutzt haben, zeigt A1 nunmehr auf die nächste Stelle in unserem BCD-Bereich. Nach Abschluß der Ausführung ist der Inhalt von A1 um acht größer als bei Beginn, und in dem vorgesehenen Bereich liegt die BCD-Codierung der eingelesenen Zahl. Diese Zahl (max. 10^{16}) kann wesentlich größer als eine 32 Bit lange Binärzahl (max. 2^{32}, etwas $4{,}3 * 10^9$) sein.

Das folgende Programmstück verwendet zur Addition von Dezimalzahlen den Befehl ABCD, so wie wir in früheren Beispielen Binärzahlen mit ADD addierten.

```
SIZE    EQU     8               Byteanzahl zur Darstellung einer
*                               Zahl
COUNT   LINK    A0,#-SIZE*2     Platz für 2 Zahlen reservieren
        LEA.L   -SIZE(A0),A1    A1 zeigt auf den Anfang des
*                               zweiten Bereichs

* Ersten Bereich löschen

CNT1    CLR.B   -(A1)           Ein Byte löschen
        CMPA.L  A7,A1           Erster Bereich gelöscht ?
        BNE.S   CNT1            Nein → Sprung
        MOVE.W  #4,D5           5 Zahlen addieren

* Summationsschleife
```

```
CNT2    LEA.L    -SIZE(A0),A1    A1 zeigt auf den Anfang des
*                                Eingabebereiches
        MOVEA.L  A1,A2           A2 zeigt hinter das Ende des
*                                Summenbereiches
        BSR.S    DRDN            Zahl einlesen

* A1 und A2 zeigen nun hinter die Enden der Dezimalzahlen

        SUB.B    D0,D0           X-Bit löschen
CNT3    ABCD     -(A1),-(A2)     2 Bytes addieren
        CMPA.L   A7,A2           Fertig mit der Addition ?
        BNE.S    CNT3            Nein → weiter
        DBRA     D5,CNT2         Nächste Zahl oder Ende
```

Im ersten Teil dieses Programms reservieren wir uns Arbeitsspeicher zur Darstellung zweier Dezimalzahlen. Wir nehmen hier an, daß unsere Zahlen maximal 8 Byte lang sind, und daß die Summe auch noch in diesen Bereich paßt. Den ersten Bereich benutzen wir für die Summe, der zweite enthält den aktuellen Summanden.

Mit der ersten kleinen Schleife löschen wir einfach den Summenbereich, indem wir ihn mit Nullen füllen. Als Schleifenabbruchbedingung nutzen wir hier, daß A7 die Grenze unseres Arbeitsspeichers markiert. Die ersten beiden Befehle in der Hauptschleife setzen A1 auf den Anfang des Bereiches für die einzulesende Zahl und A2 hinter den Summenbereich. (Diese beiden Stellen sind zwar identisch, haben aber für A1 und A2 verschiedene Bedeutung.) Nach dem Einlesen der Zahl durch Aufruf von DRDN zeigt A1 hinter den Eingabebereich.

In der nächsten Schleife addieren wir die eingelesene Zahl mittles Dezimalatrithmetik zu der bisherigen Summe. Der Zustand des X-Bits ist hierbei von großer Bedeutung, da ABCD die Summe aus seinen Operanden und dem X-Bit bildet. Es wird gesetzt, wenn ein dezimaler Übertrag auftritt, so daß die Folge der Byte-Additionen korrekt abläuft. Vor dem Schleifenbeginn müssen wir daher das X-Bit löschen. Daher ziehen wir einfach den Inhalt des unteren Bytes von D0 von sich selbst ab. Da dabei garantiert kein Borrow auftritt, wird das X-Bit gelöscht. Wir werden später noch eine elegantere Methode kennenlernen, die Bits des Statusregisters zu beeinflussen.

Bei jedem Schleifendurchlauf addieren wir zwei Bytes (mit je zwei Ziffern) der Dezimalzahlen unter Berücksichtigung des X-Bits. Tritt dabei ein dezimaler Übertrag auf (Summe > 99), so wird dies im nächsten Durchlauf durch den Zustand von C- und X-Bit angezeigt und ausgewertet. Das Z-Bit wird gelöscht, wenn eins der auftretenden Bytes von Null verschieden ist, wird aber nicht verändert, wenn das Ergebnis Null lautet. Wir haben das Z-Bit durch die Subtraktion zu Beginn der Schleife gesetzt, so daß es nach abgeschlossener Addition aller Zahlen nur dann gesetzt ist, wenn die Gesamtsumme gleich Null ist. Der Zustand der beiden anderen Condition-Codes (V und N) ist nach Ausführung eines ABCD undefiniert. Das X-Bit wird durch den CMPA-Befehl, mit dem wir auf das Ende der Schleife prüfen, nicht beeinflußt.

Sie haben sicher bemerkt, daß wir beim ABCD-Befehl die Operanden im Predecrementmodus spezifiziert haben. Dies ist die eine der beiden erlaubten Adressierungsarten; die andere ist (wie von ADDX her bereits bekannt) die Verwendung zweier Datenregister. Der Postdekrementmodus ist nötig, da die Ziffern einer Dezimalzahl immer von hinten nach vorne addiert werden müssen. Die Befehlsversion, die Datenregister als Operanden verwendet, erlaubt es nur, einzelne Bytes in Registern abzulegen und zu addieren; der Befehl hat immer Byteformat.

Der Befehl SBCD arbeitet sehr ähnlich. Auch er kennt nur diese beiden Adressierungsarten; das Zielbyte enthält nach Ausführung die Differenz zwischen dem ursprünglich im Zielbyte vorliegenden Dezimalwert und dem im Quellbyte enthaltenen Wert plus X-Bit. X- und C-Bit zeigen hier ein Borrow an, die übrigen Condition-Codes werden wie bei ABCD gesetzt.

Der letzte Befehl zur Dezimalarithmetik ist NBCD, der zum negieren dezimaler Werte benutzt wird. Er arbeitet ähnlich wie der NEGX-Befehl: Das Zielbyte wird negiert, anschließend wird noch das X-Bit abgezogen. Die Condition-Codes werden wie bei SBCD gesetzt. Anders als bei ABCD und SBCD kann jede Adressierungart vom Typ 'data-alterable' zur Spezifikation des Operanden von NBCD verwendet werden.

Obwohl es sich bei ihm eigentlich nicht um eine dezimale Operation handelt, können wir den bereits bekannten Befehl CMPM zur Behandlung dezimaler Werte nutzbringend heranziehen. Er kann nur mit im Postincrementmodus adressierten Operanden angegeben werden, ist aber nützlich zum Testen zweier im Speicher vorliegender Dezimalzahlen auf Gleichheit.

Abschließend benötigen wir noch eine Möglichkeit, unsere im BCD-Format abgespeicherten Zahlen auszugeben. Dieses Unterprogramm ist das Gegenstück zu der bereits beschriebenen Routine DRDN. Sie erhält in A1 einen Zeiger auf den Anfang des BCD-Bereiches übergeben; zur Ausgabe ruft sie unser altbekanntes Unterprogramm WRCH auf. A1 zeigt anschließend unmittelbar hinter den Bereich. Diese einfache Version gibt alle führenden Nullen mit aus; eine etwas intelligentere Version würde diese in Leerzeichen umwandeln.

```
DWRN    MOVEM.L  D0-D2,-(SP)    Register retten
        MOVEQ    #7,D1          Zähler initialisieren
        MOVEQ    #12,D2         Wert für Verschiebung
DWRN1   MOVE.L   #$30300,D0     D0 initialisieren
        MOVE.B   (A1)+,D0       Zwei Dezimalziffern
        ROR.W    #4,D0          Nibbles verschieben
        BSR      WRCH           Erste Ziffer ausgeben
        LSR.L    D2,D0          Zweite Ziffer runterschieben
        BSR      WRCH           Zweite Ziffer ausgeben
        DBRA     D1,DRWN1       Nächste Ziffern oder Ende
        MOVEM.L  (SP)+,D0-D2    Register wiederherstellen
        RTS                     und zurück
```

In der Hauptschleife dieser Routine setzen wir zunächst D0 auf den Wert $30300; wir werden gleich sehen, warum. Als nächstes holen wir uns aus dem BCD-Bereich ein Byte mit den nächsten beiden auszugebenden Ziffern und legen dieses als unterstes Byte in D0 ab. Hat es beispielsweise den Wert $56, so enthält D0 nunmehr den Wert $30356.

Der Befehl ROR.W shiftet dann das untere Wort zyklisch um 4 Positionen nach rechts, wodurch das unterste Nibble an den Anfang des Wortes gelangt. D0 enthält in unserem Beispiel nunmehr den Wert $36035. Das untere Byte enthält nun den Wert $35, den ASCII-Code des Zeichens '5', welches wir dann mittels WRCH ausgeben.

Der LSR.L-Befehl soll nun den gesamten Inhalt von D0 um 12 Positionen nach rechts shiften. Da die Anzahl der Shifts nur dann als Direktoperand spezifiziert werden kann, wenn sie nicht größer als acht ist, benutzen wir D2, welches wir vorher mit dem Wert 12 geladen haben. Bei Ausführung bringt LSR.L in unserem Beispiel den Wert $36 in das untere Byte von D0. Durch Aufruf von WRCH geben wir dann das Zeichen '6' aus. Anschließend springen wir wieder zum Anfang bzw. beenden die Schleife, falls wir alle Ziffern ausgegeben haben.

Die zur Konvertierung von der Zeichen- in die BCD-Darstellung notwendigen Verrenkungen sind noch harmlos, verglichen mit denen zur Umwandlung in die binäre Darstellung und wieder zurück. Bei vielen Prozessoren gibt es jedoch Befehle, die etwa PACK und UNPACK heißen und diese Umwandlungen mit einem Befehl erledigen. In den ersten Dokumentationen zum 68000 wurden diese Befehle auch noch erwähnt, und im 68020 sollen sie nun implementiert werden.

6
Logische Operationen

In den bisherigen Kapiteln haben wir gesehen, wie in Registern und in Speicherbereichen vorliegende Bitmuster als Zahlen oder Zeichen betrachtet und entsprechend verarbeitet wurden. Die binäre Darstellung wird in Computern bevorzugt, da es wesentlich einfacher ist, zuverlässige Schaltkreise mit nur zwei Zuständen (z.B. 'an' und 'aus') zu bauen als solche mit mehreren. Zwar ist die Kenntnis der internen Darstellung von Zeichen und Zahlen im allgemeinen uninteressant, jedoch ist es manchmal ganz sinnvoll, die Vorteile der binäre Darstellung zu nutzen und Werte nur als Ansammlung von Bits zu betrachten und zu bearbeiten. Derartige Manipulationen werden **logische** Operationen genannt, um sie von den Rechenoperationen zu unterscheiden. Einem einzelnen Bit wird dabei anstatt 1 oder 0 der logische Wert **wahr** oder **falsch** zugeordnet.

Die wohl einfachste der logischen Operationen besteht darin, jedes Bit eines Operanden zu invertieren.

```
NOT.L     D3
```

löscht jedes gesetzte Bit bzw. setzt jedes gelöschte. Wie alle anderen logischen Operationen kann NOT nur auf Operanden vom Typ 'data-alterable' angewendet werden. Dies stärkt wieder einmal die Konvention, daß Adreßregister für Adressen gedacht sind, und logische Operationen auf Adressen sind eher symptomatisch für falsche oder schludrige Programme. Es gibt jedoch auch Fälle, in denen Bitmanipulationen mit Adressen durchaus sinnvoll sind, wie wir in der Speicherzuweisungsroutine später in diesem Kapitel noch sehen werden.

Mehrere logische Operationen verwenden zwei Operanden. Sie unterscheiden sich von den arithmetischen Operationen dadurch, daß die einzelnen Bits der Operanden paarweise verknüpft werden, anstatt das Byte, Wort oder Vollwort als einzelnen Wert zu behandeln. Die Ausführung logischer Operationen entspricht der parallelen Ausführung von 8, 16 oder 32 Ein-Bit-Rechnungen.

Der Befehl OR erzeugt ein Resultat, in dem jedes Bit gesetzt ist, das im Quell- **oder** im Zieloperanden gesetzt war. Enthält beispielsweise das untere Byte von D1 das Bitmuster 11001100, das von D2 das Muster 11100001, so ergibt

```
OR.B    D1,D2
```

das Muster 11101101 im unteren Byte des Zieloperanden. Eine andere Ansicht der Wirkung von OR ist, daß ORen (oder 'ODERn', wie die Programmierer gerne sagen) mit 0 ein Bit nicht beeinflußt, ORen mit 1 es jedoch immer auf 1 setzt. Wir können also beispielsweise die Bits des oberen Worts eines Registers auf 1 setzen, ohne die untere Hälfte zu verändern:

```
OR.L    #$FFFF0000,D3
```

Eine ganz andere Art der Kombination von Bits besteht darin, sie zu ANDen (ebenfalls so ein merkwürdiges Wort). Die Bits des Resultats eines ANDs sind nur dann gesetzt, wenn **sowohl** im Quell- **als auch** im Zieloperanden die entsprechenden Bits gesetzt waren. Mit denselben Operanden wie oben in D1 und D2 erhalten wir als Resultat von

```
AND.B   D1,D2
```

das Ergebnis 11000000 im unteren Byte von D2. AND ist das Gegenteil von OR in dem Sinne, daß ANDen mit 0 eine 0 ergibt, ANDen mit 1 jedoch den Wert eines Bits nicht ändert. Der Befehl AND ist sehr nützlich, um einen Wert teilweise zu **maskieren**, d.h. unerwünschte Bits zu löschen. Nehmen wir beispielsweise an, wir haben nach einer Berechnung ein 8-Bit-Ergebnis im unteren Byte von D4 erhalten, wissen aber nichts über den Zustand der restlichen Bits dieses Registers. Wir können nun die unnötigen Bits löschen, so daß D4 nur noch unser Ergebnis enthält:

```
AND.L   #$000000FF,D4
```

Als dritten Befehl zur Verknüpfung zweier Bitmuster wollen wir das exklusive OR (EOR) vorstellen. Dieses ergibt eine 1 in einem Resultat, wenn **entweder** das entsprechende Bit im Quell- **oder** das im Zieloperanden gesetzt war. Die Betonung liegt hier auf dem **entweder**; die beiden Bits müssen verschiedene Werte haben, damit das Bit im Resultat gesetzt ist. Sind sie beide gesetzt oder beide gelöscht, so ist das Ergebnis 0. Enthalten D1 und D2 wieder die obigen Werte, so ergibt

```
EOR.B   D1,D2
```

das Ergebnis 00101101 in D2.

Eine andere Sichtweise ist die, daß EORen mit 0 ein Bit unverändert läßt, während EORen mit 1 ein Bit invertiert. EORen mit 1 entspricht somit der Ausführung von NOT mit dem anderen Operanden.

Die für die Operanden von EOR erlaubten Adressierungsarten unterschieden sich leicht von denen für AND und OR: Der Quelloperand muß ein Datenregister sein. AND und OR müssen zwar auch mindestens ein Datenregister unter ihren Operanden

haben, jedoch kann dies auch der Zieloperand sein. Alle drei Befehle haben eine Form für Direktoperanden (ANDI, ORI und EORI), die benutzt werden kann, wenn der Quelloperand ein festes Bitmuster ist (natürlich kann auch beim einfachen AND und OR ein Direktoperand angegeben werden). Eine exklusive Eigenschaft dieser Befehle liegt darin, daß sie als Zieloperanden das Prozessorstatusregister (SR) verwenden können. Wird der Befehl dabei mit Bytegröße angegeben, so wird nur das untere Byte des Statusregisters, das Condition-Code-Register (CCR), beeinflußt. Bei Wortformat wird das ganze Statusregister beeinflußt, daher zählt der Befehl in diesem Fall zu den privilegierten und kann nur im Supervisormodus angewandt werden. Diese Formen der Befehle dienen dazu, bestimmte Bits des Status und der Condition-Codes zu setzen (ORI), zu löschen (ANDI) oder zu invertieren (EORI), ohne die anderen Bits zu beeinflussen. Um das C-Bit zu löschen, können wir den Befehl

```
ANDI.B    #$FE,CCR
```

benutzen, und um den Tracemodus zu setzen, den Befehl

```
ORI.W     #$8000,SR
```

6.1 SHIFTEN UND ROTIEREN

Wir haben jetzt gesehen, wie wir einzelne Bits an ihrer Position behandeln können. Nunmehr wollen wir Ihnen zeigen, wie Bitmuster innerhalb von Registern oder Speicherbereichen verschoben werden können. Dafür gibt es vier verschiedene Befehle, und jeder von ihnen hat zwei Versionen: die eine für Bewegungen nach links, die andere für solche nach rechts. Alle **Shift-Befehle** verwenden drei Arten von Operanden. Liegt der Operand im Speicher, so hat der Befehl automatisch Wortformat und es wird nur um eine Position geshiftet. Liegt der Operand dagegen in einem Datenregister vor, so sind alle drei Größenangaben erlaubt, und die Anzahl der Shifts kann entweder als Direktoperand zwischen 1 und 8 oder durch den Inhalt eines anderen Datenregisters angegeben werden.

Der Befehl LSL (logical shift left) schiebt die Bits seines Operanden nach links, wobei er die rechts freiwerdenden Stellen mit Nullen auffüllt. Enthält das durch A1 adressierte Wort im Speicher das Bitmuster 1011111111111111, so besteht der Effekt der Ausführung von

```
LSL.W     (A1)
```

darin, daß dieses Wort den Wert 0111111111111110 erhält. Das C- und das X-Bit erhalten den Wert des verlorengegangenen Bits, und das N- und das Z-Bit werden wie gehabt in Abhängigkeit vom Ergebnis gesetzt.

Wie oben erwähnt, kann ein in einem Datenregister liegendes Bitmuster mit einem Befehl um mehr als eine Stelle geshiftet werden. Die Anzahl der Stellen kann auf zwei Arten angegeben werden. Ein konstanter Shift um 1 bis 8 Stellen kann durch einen Direktoperanden spezifiziert werden, z.B.:

```
LSL.L    #4,D2
```

Alternativ dazu kann als Shiftwert der Inhalt eines anderen Datenregisters verwendet werden:

```
LSL.L    D1,D2
```

Der verwendete Wert ist dabei der Inhalt des Registers modulo 64.

Der Befehl LSR führt einen logischen Shift nach rechts aus. Er füllt dabei die links freiwerdenden Stellen mit Nullen auf; das C- und das X-Bit erhalten den Wert des zuletzt herausgeschobenen Bits.

Shiften wir eine Binärzahl eine Stelle nach links, so entspricht das einer Multiplikation mit zwei (ohne Berücksichtigung eines evtl. Überlaufs), shiften wir sie eine Stelle nach rechts, so entspricht dies einer Division durch zwei (ohne Beachtung des Rests). Wir können daher das Shiften zum Multiplizieren und Dividieren mit Zweierpotenzen verwenden. Bei Verwendung von LSR zum Dividieren dürfen allerdings keine negativen Dividenden benutzt werden, da durch das Auffüllen mit Nullen das Vorzeichen verloren geht.

Zur Überwindung dieses Problem gibt es zwei weitere Shift-Befehle: die **arithmetischen Shifts** ASL und ASR, die davon ausgehen, daß ein Operand eine Zahl in Zweierkomplement-Darstellung ist. Die möglichen Operanden sind die gleichen wie bei LSL und LSR; sie unterscheiden sich aber von diesen Befehlen durch die Behandlung des Vorzeichens und der Condition-Codes. Bei Shifts nach rechts ist der Unterschied kaum merkbar: LSL löscht generell das V-Bit, wogegen ASL es setzt, wenn ein arithmetischer Überlauf auftrat und es anderenfalls löscht; das sich nach Ausführung des Befehls ergebende Bitmuster ist bei beiden Befehlen identisch.

Die unterschiedliche Behandlung des Vorzeichens kommt nur bei Shifts nach rechts zum Tragen. Die links aufgefüllten Bits sind Kopien des ursprünglichen Vorzeichen-Bits, wodurch ASR eine Division sowohl für positive als auch für negative Zahlen ermöglicht. Bei positiven Zahlen haben LSR und ASR die gleiche Wirkung, bei negativen Zahlen jedoch füllt ASR die freiwerdenden Positionen mit Einsen, anstatt wie LSR mit Nullen. Enthält beispielsweise das untere Byte von D1 das Bitmuster 11101100 (= −20 dezimal), so bewirkt die Ausführung von

```
ASR.B    #2,D1
```

daß dieses Byte auf 11111011 (= −5 dezimal) gesetzt wird. Nachkommastellen gehen bei dieser Division natürlich verloren, so daß das Ergebnis einer Division in jedem Fall abgerundet wird:

5 um eine Stelle nach rechts geshiftet ergibt 5 / 2 = 2
−5 um eine Stelle nach rechts geshiftet ergibt (−5) / 2 = −3

Sowohl beim arithmetischen als auch beim logischen Shiften gehen die aus dem Operanden 'herausgeschobenen' Bits verloren. Dies läßt sich durch die Verwendung von Rotationsoperationen, die auch als **zyklische Shifts** bezeichnet werden, vermeiden. Bei diesen werden die an einem Ende des Operanden 'herausgeschobenen' Bits am anderen Ende wieder eingefügt. Dadurch geht keinerlei Information verloren, und durch eine entsprechende Anzahl von Shifts (z.B. 8 bei einem Byte) kann der alte Inhalt der Operanden wiederhergestellt werden.

Es gibt zwei verschiedene Formen des Rotationsbefehls, die beide wieder eine 'linke' und eine 'rechte' Version kennen. Die Operanden können wie bei den anderen Shift-Operationen angegeben werden. ROL und ROR rotieren die Operanden um die angegebene Anzahl von Positionen und hinterlassen im C-Bit eine Kopie des letzten Bits, das von einem Ende zum anderen transportiert wurde. N- und Z-Bit werden wie üblich abhängig vom Ergebnis gesetzt, das V-Bit wird immer gelöscht, und das X-Bit bleibt unbeeinflußt. ROXL und ROXR verhalten sich sehr ähnlich, jedoch wandert jedes herausgeschobene Bit in das X-Bit, und das eingeschobene Bit wird vorher aus dem X-Bit kopiert. Diese Befehle benötigen einen Shift mehr als die normalen Rotierbefehle um den Operanden wieder in den ursprünglichen Zustand zu bringen. Ihre Bedeutung liegt darin, daß sie die einzigen Shift-Operationen sind, die es gestatten, das Ergebnis eines früheren Befehls als in den Operanden hereinzuschiebendes Bit zu benutzen. Sie können daher auch verwendet werden, um Objekte mit mehr als 32 Bits zu shiften. Nehmen wir beispielsweise an, wir hätten in D1 und D2 einen 64-Bit-Wert, so können wir diesen wie folgt nach links shiften:

```
        LSL.L   #1,D2       Untere Hälfte: das herausgeschobene
*                           Bit liegt als X-Bit vor
        ROXL.L  #1,D1       Obere Hälfte: das X-Bit wird
*                           hereingeschoben
```

6.2 UMWANDLUNG EINER HEXADEZIMALZAHL IN IHRE ZEICHENDARSTELLUNG

Mit Hilfe der bisher bekannten logischen Operationen können wir ein Programmstück schreiben, das eine in einem Register abgespeicherte Hexadezimalzahl in ihre Zeichendarstellung konvertiert. Jede hexadezimale Ziffer dieser Zahl besteht aus 4 Bits, also einem Nibble. Durch Rotation bringen wir jedes Nibble an die untere Position

des Registers und maskieren es dort mit einem AND-Befehl. So erhalten wir eine Zahl zwischen 0 und 15, die wir dann als Offset in einer Tabelle mit den entsprechenden Zeichenkonstanten ausnutzen:

```
* Die umzuwandelnde Zahl steht in D1. A0 zeigt auf einen 8 Byte
* langen Puffer, in dem die Zeichendarstellung abgelegt wird.

          MOVEQ     #7,D0       Schleifenzähler

LOOP      ROL.L     #4,D1       Um ein Nibble nach links schieben
          MOVE.B    D1,D2       Kopieren der unteren Nibbles
          ANDI.L    #$0F,D2     Unteres Nibble maskieren
          MOVEA.L   D2,A1       Offset muß aufgrund der gewünschten
*                               Adressierungsart in einem
*                               Adreßregister stehen
* Das richtige Zeichen im Puffer ablegen

          MOVE.B    CHARTAB(A1),(A0)+
          DBRA      D0,LOOP     Nächstes Nibble oder Ende

* Aussprung; Umwandlung beendet

          ....
          ....

CHARTAB   DC.B      '0123456789ABCDEF'
```

6.3 OPERATIONEN AUF EINZELNEN BITS

Zur Behandlung einzelner Bits in einem Datenregister oder einer Speicherstelle gibt es fünf Befehle. Einer von ihnen, TAS, ist ziemlich speziell und wird daher erst später behandelt. Die anderen vier, BTST, BCLR, BSET und BCHG, bilden eine Familie, in der alle Befehle dieselben Operandenformate haben. Jeder dieser vier operiert auf einem einzelnen Bit, dessen Lage durch seine Nummer im Speicherbyte bzw. Register spezifiziert wird. Bits werden ab der rechten (unteren) Stelle mit 0 beginnend aufwärts gezählt. Die Bits eines Registers haben dann also (von links nach rechts) die Nummern 31 bis 0, die eines Bytes im Speicher die Nummern 7 bis 0.

Der Befehl BTST testet das Bit nur; ist es gesetzt, so wird das Z-Bit gelöscht, anderenfalls gesetzt. Alle anderen Condition-Codes werden nicht verändert. Die Position des Bits kann entweder durch einen Direktoperanden oder den Inhalt eines Datenregisters angegeben werden. Der Zieloperand kann in jeder Adressierungsart vom Typ 'data' spezifiziert werden, aber natürlich nicht als Direktoperand. Ist der Quelloperand ein Datenregister, so wird die Bitnummerierung modulo 32 gehandhabt. Die beiden Befehle

```
          BTST      #3,D7
          BTST      #35,D7
```

prüfen also beide Bit 3 von D7. Ist der Quelloperand ein Byte im Speicher, so wird modulo 8 gerechnet.

Die drei anderen Befehle dieser Familie prüfen auch das angegebene Bit, können es aber auch verändern. Ihr Zieloperand muß daher auch vom Typ 'data-alterable' sein. BCLR löscht das Bit, BSET setzt es, und BCHG ändert seinen Wert.

Da der Speicher byteweise organisiert ist, muß eine Bitoperation zunächst ein ganzes Byte lesen, das entsprechende Bit verarbeiten und dann das Byte wieder in den Speicher zurückschreiben. Bei Anwendung von BCLR, BSET oder BCHG zur Veränderung von Bits in im Speicher liegenden Kontrollregistern peripherer Geräte können daher unerwartete Effekte auftreten, da oft schon das Lesen eines Kontrollregisters irgendwelche Aktionen des Gerätes hervorrufen kann. In diesem Fall ist es sicherer, das gewünschte Bitmuster komplett zu erstellen und es anschließend mit MOVE im Kontrollregister abzulegen.

Der Befehl TAS (test and set) arbeitet auch mit nur einem Bit als Operanden. Er ist weniger flexibel als die anderen Bitbefehle, da er nur im Byteformat verwendbar ist; es ist dabei immer Bit 7, das er zunächst testet und anschließend setzt. Das N- und das Z-Bit werden entsprechend dem ursprünglichen Inhalt des spezifizierten Bytes gesetzt oder gelöscht. TAS kann zwar mit einem beliebigen Operanden vom Typ 'data-alterable' angegeben werden, aber seine große Bedeutung liegt in der Art, wie er ein Byte im Speicher anspricht. Er verwendet dazu etwas, was man als 'lesen-modifizieren-schreiben'-Speicherzugriff bezeichnet. Dies bedeutet, daß TAS die Kontrolle über den Speicher während der ganzen Zeit seiner Abarbeitung nicht abgibt, so daß nichts und niemand das Operandenbyte lesen oder verändern kann, solange TAS es behandelt. Wie bereits in Kapitel 1 erwähnt, ist dies eine lebenswichtige Funktion in einem System, in dem mehrere Computer den gleichen Hauptspeicher benutzen. Nur so wird die Verwendung von Markierungen im Speicher möglich, die angeben, ob bestimmte gemeinsame Betriebsmittel momentan belegt sind oder nicht. Solche Markierungen werden auch als **Semaphore** bezeichnet.

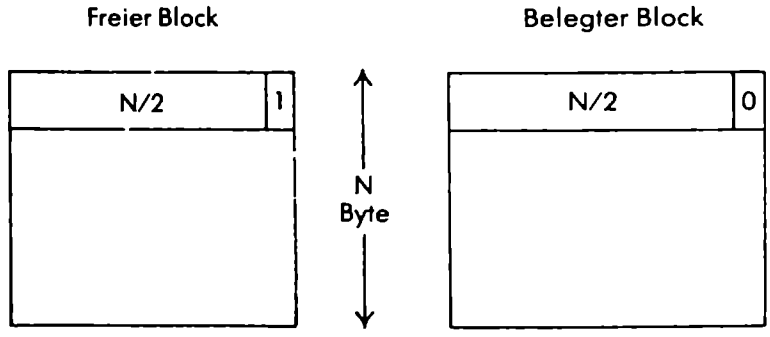
Semaphore werden auch benötigt, wenn in einem Computer mehrere Programme mehr oder weniger unabhängig voneinander nebenläufig abgearbeitet werden. Viele Betriebssysteme erlauben es, mehrere Programme gleichzeitig ablaufen zu lassen, und sorgen dafür, daß die Prozessorzeit zwischen diesen aufgeteilt wird. Selbst in einfacheren Systemen laufen die Interruptroutinen (siehe Kapitel 7) zu beliebigen Zeiten während der Abarbeitung des Hauptprogramms. In einem einzelnen Computer können BCLR, BSET und BCHG ebenso gut wie TAS zur Behandlung der Semaphore verwendet werden, da auch sie eine 'test and set'-Operation in einem einzigen Befehl ausführen. Ein Programm kann daher kein Semaphor verändern, solange ein anderes dieses gerade bearbeitet.

6.4 EIN SPEICHERVERWALTUNGSPAKET

Bei einfachen Programmen ist der Bedarf an Speicher für die verschiedenen Daten, Puffer etc. normalerweise vorhersagbar; der vorhandene Speicher kann daher dementsprechend aufgeteilt werden. Viele Programme jedoch müssen in der Lage sein, den Speicher für verschiedene Zwecke aufzuspalten, und die für bestimmte Zwecke tatsächlich benötigte Speichermenge stellt sich erst während des Programmablaufes heraus. In diesen Fällen ist es recht praktisch, Speicherbereiche reservieren und nach Gebrauch wieder freigeben zu können. Dieser Mechanismus kann durch eine Sammlung von Unterprogrammen realisiert werden, die dann als **Speicherverwaltungspaket** bezeichnet wird.

Ein einfaches Paket enthält nur zwei Routinen: eine zur Anforderung von Speicher und eine zur Freigabe. Der Speicher selbst kann dabei wie folgt organisiert sein: Liegt der Speicher als durchgehender Bereich vor, so läßt er sich in zusammenhängende Blöcke unterteilen. Jeder dieser Blöcke besitzt am Anfang ein sogenanntes Kopfwort, in dem die Länge des Blocks sowie eine Markierung, ob er frei oder belegt ist, untergebracht ist. Das Kopfwort des allerletzten Blocks enthält nur eine Null.

Wir wollen es zur Regel machen, daß die Länge eines jeden Blocks (in Byte) ein Vielfaches von vier ist; die Kopfworte sind Vollworte. So kann ein Block den gesamten in einem 68000-System denkbaren Speicher enthalten, und wir werden zwischen den Blöcken auch keine Löcher erhalten, die selbst für ein Kopfwort zu klein sind. Um die Kopfworte mit Vollwortbefehlen bearbeiten zu können, muß jeder Block an einer geradzahligen Adresse beginnen. Da die Blocklängen stets Vielfache von vier sind, sind die beiden unteren Bits im Kopfwort immer gleich Null. Wir können daher das untere Bit für die Angabe, ob ein Block frei oder belegt ist, verwenden.



Die Größe jedes dieser beiden Blöcke (inklusive Kopfwort) beträgt N Byte.

Das Unterprogramm zur Belegung von Speicher nennen wir GETBLK. Es verwendet einen sogenannten 'first-fit'-Algorithmus, d.h. es arbeitet sich durch die Blöcke durch und belegt den ersten genügend großen freien Block. Werden bei diesem Durchsu-

chen hintereinanderliegende freie Blöcke entdeckt, so werden diese zu einem Block verschmolzen. Wenn nach einer Weile viele Blöcke benutzt und wieder freigegeben wurden, ist er sehr wahrscheinlich, daß kleinere Blöcke hintereinander liegen. Es wäre töricht, GETBLK scheitern zu lassen, nur weil kein einzelner Block groß genug ist, aber es ist auch nicht nötig, dieses Zusammenschmelzen durchzuführen, bevor es wirklich notwendig ist.

```
* Routine zur Belegung von Speicherbereichen
*
* Bei Einsprung enthält D1 die Anzahl der benötigten Bytes
*
* Bei Aussprung enthält A0 die Adresse des ersten Bytes des
* belegten Blocks, oder 0, falls die Routine scheiterte.
* D0 enthält einen Fehlercode mit der folgenden Bedeutung:
*
* 0 : Block belegt* 1 : kein Speicher mehr frei
* 2 : Verkettung der Blöcke fehlerhaft

GETBLK   MOVEM.L  A1/D1-D3,-(SP) Arbeitsregister retten
         ADDQ.L   #3,D1          Anzahl der benötigten Bytes um drei
*                                erhöhen
         ANDI.B   #$FC,D1        Auf ein Vielfaches von 4 abrunden
         ADDQ.L   #4,D1          Blockgröße = (gerundete) Größe + 4
         BLE      GBC7           Negativer Wert → Fehler

         GBCRTY   LEA.L          BLKLIST,A0 Adresse des ersten Blocks

* Durchsuchen der Liste nach einem freien Block. Verschmelzen
* benachbarter freier Blöcke.

GBC1     MOVE.L   (A0),D2        D2 = Kopfwort des Blocks
         BLE.S    GBC6           Ende der Liste (oder Fehler)
         BCLR.L   #0,D2          Testen und Löschen der Markierung
         BNE.S    GBC2           Freier Block → Größe prüfen
         ADDA.L   D2,A0          A0 = Adresse des nächsten Blocks
         BRA.S    GBC1           Weitersuchen

* Freier Block gefunden -> Größe testen, evtl. Verschmelzen.

GBC2     MOVE.L   A0,D3          D3 = Adresse des Blocks
*
GBC3     ADDA.L   D2,A0          A0 = Adresse des nächsten Blocks
         MOVE.L   (A0),D2        D2 = Kopfwort des nächsten Blocks
         BMI.S    ERRSTORE       Kein Kopfwort → Fehler in der
*                                Verkettung
         BCLR.L   #0,D2          Markierung testen und löschen
         BNE.S    GBC3           Block frei → Weitersuchen

* An dieser Stelle gilt:
*            D1 = benötigte Größe in Byte
*            D3 = Startadresse des freien Bereichs
*            A0 = Endadresse des Bereichs + 1

* Verschmelzen der freien Bereiche
```

```
GBC4    MOVE.L   A0,D2       Endadresse kopieren
        SUB.L    D3,D2       D2 = Länge des verschmolzenen
*                            Bereichs
        BSET     #0,D2       "Frei"-Markierung setzen
        MOVEA.L  D3,A1       Startadresse in ein Adreßregister
        MOVE.L   D2,(A1)     Kopfwort des verschmolzenen Bereichs
        BCLR     #0,D2       Markierung zur Berechnung löschen
*
        SUB.L    D1,D2       Aufspalten des Blocks
*                            D2 = Größe des Rests
        BLT.S    GBC1        Block zu klein → Weitersuchen
        BEQ.S    GBC5        Block paßt genau → Belegen

* Der Rest des Blockes muß nun als neuer Block markiert werden.

        SUBA.L   D2,A0       A0 = Adresse des Rests
        BSET     #0,D2       D2 = Kopfwort des Rests
        MOVE.L   D2,(A0)     Rest als neuer Block gekennzeichnet
*
GBC5    MOVE.L   D1,(A1)     Größe abspeichern (Markierung = 0)
        ADDQ.L   #4,D3       D3 = Adresse des belegten Bereichs
        MOVEA.L  D3,A0       Im Ergebnisregister ablegen
        CLR.L    D0          Keine Fehler
        BRA.S    GBEXIT      Rücksprung

* Fehler oder Ende der Liste erreicht

GBC6    BMI.S    ERRSTORE    Fehler in der Verkettung

* Nicht genügend Platz frei

GBC7    MOVEQ    #1,D0       Fehlercode
GBERREX SUBA.L   A0,A0       A0 = 0
GBEXIT  MOVEM.L  (SP)+,A1/D1-D3
        RTS

* Fehler in der Verkettung

ERRSTORE MOVEQ   #2,D0       Verkettung der Blöcke fehlerhaft
        BRA.S    GBERREX     Fehlerausgang
```

Das Unterprogramm zur Freigabe eines reservierten Blocks heißt FREEBLK. Um den Block als frei zu markieren, muß nur das untere Bit in seinem Kopfwort gesetzt werden. Zusätzlich kann jedoch auch noch überprüft werden, ob die zwecks Freigabe übergebene Adresse wirklich die eines Blockanfangs ist. Wir testen daher, ob die Adresse gerade und damit auch, ob das untere Bit wirklich gelöscht ist. Ein häufig auftretender Programmierfehler besteht darin, einen Block mehrmals freizugeben, was durch diesen einfachen Test verhindert wird. Die mehrfache Freigabe eines Blockes kann fatale Folgen haben, wenn dieser bereits für einen anderen Zweck wieder belegt wurde.

FREEBLK überprüft außerdem, daß sowohl die Adresse als auch das Kopfwort im obersten Byte den Wert Null stehen haben, da es sich bei dem Vollwort sonst nicht um eine Adresse bzw. eine Längenangabe handeln kann. Den mühsamen und zeitrau-

benden Prozeß des Durchsuchens der Blockliste, um sicherzustellen, daß die Adresse tatsächlich einen belegten Block bezeichnet, haben wir uns erspart.

FREEBLK springt sofort zurück, wenn es eine Null als Argument erhält. Dadurch stellen wir sicher, daß das Resultat eines Aufrufs von GETBLK stets ein gültiges Argument für FREEBLK ist, auch wenn der Aufruf von GETBLK scheiterte.

```
* Freigabe von mittels GETBLK belegten Blöcken
*
* Bei Einsprung enthält D1 die Adresse des Blocks
* Bei Aussprung ist D0 = 0, falls der Block freigegeben werden
* konnte, anderenfalls (die Adresse scheint keinen belegten Block
* zu bezeichnen) gilt D0 = 1.

FREEBLK  MOVEM.L  D1/D2/A0,-(SP)
         CLR.L    D0          Returncode setzen
         TST.L    D1          Testen der Adresse
         BEQ.S    FBEXIT      Null → Rücksprung

         SUBQ.L   #4,D1       Zeiger auf das Kopfwort des Blocks
         MOVE.L   D1,D2       Kopieren der Adresse des Kopfworts

* Untersuchung der Adressen: Sie sollten gerade sein, und das
* oberste Byte muß den Wert Null haben

         ANDI.L   #$FF000001,D2  Hat die Adresse die Gestalt
*                                0000 0000 dddd .. ddd0 ?
         BNE.S    FBERR          Nein → Fehler

         MOVEA.L  D1,A0          Kopieren in ein Adreßregister
         MOVE.L   (A0),D2        D2 = Kopfwort

         ANDI.L   #$FF000001,D2  Hat das Kopfwort die Gestalt
*                                0000 0000 dddd .. ddd0 ?
         BNE.S    FBERR          Nein → Fehler

         BSET     #0,3(A0)       Block als frei markieren
FBEXIT   MOVEM.L  (SP)+,D1/D2/A0
         RTS

FBERR    MOVEQ    #1,D0          Fehlercode
         BRA.S    FBEXIT         Zurückspringen
```

Der anfangs zur Verfügung stehende Bereich kann als ein großer Block, gefolgt von dem Wort Null, wie folgt definiert werden:

```
BLKLIST  DC.L     BLKEND-BLKLIST+1   Größe + "Frei"-Markierung
         DS.L     1000               Freier Speicher
BLKEND   DC.L     0                  Schlußmarkierung
```

7
Behandlung von Exceptions

In allen bisher behandelten Beispielprogrammen wurde die Adresse des nächsten auszuführenden Befehls implizit oder explizit durch den vorhergehenden bestimmt. Normalerweise ist es der in der Befehlsfolge nächste, aber durch eine Verzweigung, einen Sprung oder einen Rücksprung kann die Fortsetzung an einer anderen Stelle erfolgen.

Dieses Kapitel behandelt die Fälle, in denen der Programmfluß in anderer Weise gesteuert wird; diese Fälle nennt man **Exceptions** (Ausnahmebedingungen). Sie werden hauptsächlich aus zwei Gründen verwendet. Zum einen ist durch sie die Möglichkeit gegeben, auf bestimmte Ereignisse unmittelbar reagieren zu können. Derartige Ereignisse sind meist Signale von peripheren Einheiten, also etwa das Drücken einer Taste am Terminal, oder die Nachricht, daß der Papiervorrat des Druckers erschöpft ist. Zum zweiten ist es dem Computer so möglich, bestimmte interne Fehler zu melden und ein Unterprogramm zur Behandlung derselben anzuspringen. Solche Fehler sind beispielsweise illegale Befehlscodes oder das Ansprechen nicht existenter Speicherstellen.

Bei Eintritt einer Exception rettet der 68000 die Werte des Programmzeigers und des Statusregisters, dann fährt er mit der Ausführung an einer durch den sogenannten **Exceptionvektor** adressierten Stelle fort. Dieser liegt im Speicher ziemlich weit unten, hat also eine niedrige Adresse. Da Programmzähler und Statusregister gerettet werden, kann die Ausführung anschließend an der Stelle wieder aufgenommen werden, wo die Exception auftrat. Die Behandlung der Exception entspricht daher der eines Unterprogrammaufrufs zwischen zwei Befehlen.

Wie bereits erwähnt können Exceptions aus zwei Ursachen auftreten: **interne**, wenn der Prozessor selbst irgendwelche Anomalien entdeckt, oder **externe**, wenn irgendein Bestandteil des Systems die Aufmerksamkeit des Prozessors erregen will. Externe Exceptions werden oft auch als **Interrupts** (Unterbrechungen), interne als **Traps** (Fallen) bezeichnet.

Im einzelnen können folgende externe Exceptions auftreten:

Interrupts
Busfehler
externer Reset (Rücksetzen)

Die internen Exceptions sind:

Illegale Befehle
Nicht implementierte Befehle
Adreßfehler
Privilegierte Befehle im Benutzermodus
Tracing Division durch Null
TRAP, TRAPV, CHK

7.1 EXCEPTIONVEKTOREN

Bei Auftritt einer Exception ruft der Prozessor einen sogenannten **Exception-Handler** auf, ein Unterprogramm, das der Benutzer zur Behandlung dieser Ausnahmebedingung angegeben hat. Die Adresse dieses Unterprogramms muß dem Prozessor auf irgendeine Weise bekannt gemacht werden. Bei einigen Prozessoren wird für jede Exception ein und dasselbe Unterprogramm angesprungen, dessen Adresse an einer bestimmten, festen Stelle im Speicher abgelegt ist; um zu erfahren, was eigentlich passiert ist, muß dieses Unterprogramm zunächst einige Systemregister auslesen.

Der 68000 verwendet eine allgemeinere Methode: Für jede Exception kann ein eigenes Unterprogramm angegeben werden. Die untersten 1024 Bytes des Speichers enthalten die Adressen dieser Unterprogramme. Jeder dieser **Exceptionvektoren** belegt vier Byte, und seine Nummer entspricht somit der durch vier geteilten Adresse. Lage und Nummer dieser Vektoren sind in der folgenden Tabelle aufgeführt.

Die Nummer des Exceptionvektors liegt für alle Traps und für alle Interrupts, die den Autovektor-Mechanismus verwenden, fest. Andere Interrupts erzeugende Schaltkreise müssen dem 68000 eine Vektornummer übergeben; die Auswahl dieser Nummer ist dem Entwickler überlassen.

Wird der Adreßbereich des 68000 durch Verwendung der drei Speicherfunktions-Ausgabeleitungen in verschiedene Bereiche unterteilt, so werden außer dem Reset-Vektor alle Vektoren im Supervisor-Datenbereich abgelegt; der Reset-Vektor hingegen ist im Supervisor-Adreßbereich zu finden.

0	0	Reset: Initialer SSP
1	4	Reset: Initialer PC
2	8	Busfehler
3	C	Adreßfehler
4	10	Illegaler Befehl
5	14	Division durch Null
6	18	CHK-Befehl
7	1C	TRAPV-Befehl
8	20	privilegierter Befehl im Benutzermodus
9	24	TRACE
10	28	Nichtimplementierter Befehl (1010)
11	2C	Nichtimplementierter Befehl (1111)
12	30	(Nicht verwendet, aber von Motorola
:	:	für zukünftige Erweiterungen reserviert)
23	5C	
24	60	Unechter Interrupt
25	64	Interrupt-Autovektor der Stufe 1
26	68	Interrupt-Autovektor der Stufe 2
27	6C	Interrupt-Autovektor der Stufe 3
28	70	Interrupt-Autovektor der Stufe 4
29	74	Interrupt-Autovektor der Stufe 5
30	78	Interrupt-Autovektor der Stufe 6
31	7C	Interrupt-Autovektor der Stufe 7
32	80	TRAP #0
:	:	
47	BC	TRAP #15
48	C0	(Nicht verwendet, aber von Motorola
:	:	für zukünftige Erweiterungen reserviert)
63	FC	
64	100	
:	:	Vom Benutzer deklarierte
255	3FC	Interruptvektoren

7.2 BENUTZER- UND SUPERVISORMODUS

Wie bereits im ersten Kapitel erwähnt, kann der 68000 Befehle im Benutzer- oder im Supervisormodus ausführen. Diese beiden Modi unterscheiden sich bezüglich der verwendbaren Befehle: Im Benutzermodus sind nicht alle Befehle des 68000 anwendbar. Betriebssystem-Programme laufen daher normalerweise im Supervisormodus, da dann auch die sogenannten privilegierten Befehle benutzt werden dürfen. Verschiedene Schlüsselbefehle sind im Benutzermodus verboten; der Versuch, einen dieser Befehle auszuführen, führen zu einer Exception, und die Kontrolle über den Computer fällt wieder an das Betriebssystem zurück. Die privilegierten Befehle lassen sich in zwei Gruppen unterteilen: solche, die die Arbeit des Prozessors beeinflussen, wie etwa STOP und RESET, und solche, die das Programm in den Supervisormodus versetzen, ohne dabei die Ablaufkontrolle abzugeben.

Eine Ausgangsleitung des Prozessors zeigt den momentanen Modus an. Diese Leitung kann auf zwei Arten dazu verwendet werden, den dem Supervisor »gehörenden« Speicher vor dem Benutzer zu schützen. Zum einen kann bei jedem Speicherzugriff hardwaremäßig geprüft werden, ob dieser gestattet ist, und kann ggf. unterbunden werden. Zum anderen kann der im Supervisormodus zur Verfügung stehende Speicher komplett von dem für Benutzerprogramme verwendbaren getrennt werden. Die im Supervisormodus verwendete Adresse 1000 spricht also eine andere physikalische Speicherzelle an als die Adresse 1000 im Benutzermodus. Bei dieser Methode ist der Privatspeicher des Supervisormodus für den Benutzer unsichtbar.

Wie wir bereits mehrmals erwähnten, hat es mit dem Adreßregister A7 eine besondere Bewandtnis. Es wird von einigen Befehlen und bei Exceptions implizit als Stackzeiger benutzt. Der Name A7 wird eigentlich für physikalisch verschiedene Register verwendet, den Benutzer-Stackzeiger (USP) und den Supervisor-Stackzeiger (SSP). Welches dieser Register bei Benennung von A7 angesprochen wird, hängt vom jeweiligen Modus ab. Der Inhalt von SSP ist also vom Benutzerprogramm her nicht feststellbar; es gibt aber einen Befehl, um vom Supervisormodus aus den Inhalt von USP zu beeinflussen. Dies ist notwendig, da das Betriebssystem in der Lage sein muß, dieses Register lesen und setzen zu können. Dieser Befehl ist eine spezielle Version von MOVE, wobei USP als Quell- oder Zieloperand verwendet wird. Dieser Befehl ist privilegiert, aber nicht etwa aus Gründen des Schutzes, sondern weil ein Programm, daß diesen Befehl im Benutzermodus verwenden will, noch ganz andere Macken haben könnte. Hier würde schließlich die Angabe von SP ausreichen.

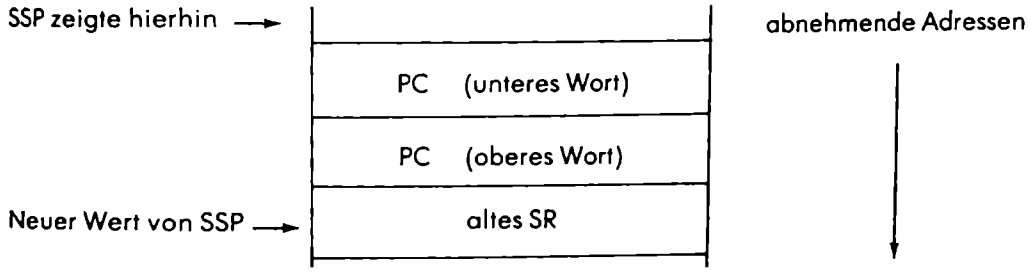
Da der Modus nur bei Auftreten einer Exception gewechselt werden kann, ist der Moduswechsel stets mit einem Sprung zu einem der in der **Vektortabelle** angegebenen Unterprogramme verbunden. Indem der Speicherbereich dieser Sprungtabelle geschützt wird, kann sichergestellt werden, daß kein Programm in den Supervisormodus geraten kann, ohne ein Systemunterprogramm anzuspringen. Somit können die von

Benutzerprogrammen ausführbaren Operationen so eingeschränkt werden, daß das Betriebssytem die Kontrolle über den Computer behält.

Es gibt verschiedene Möglichkeiten, aus dem Supervisor- wieder in den Benutzermodus zu gelangen. Z.B. können im Supervisormodus Befehle ausgeführt werden, die das Statusregister (SR) verändern, und somit kann durch einfaches Umsetzen des Modus-Bits im Statusregister der Modus geändert werden. Eine andere Methode ist die Rückkehr aus einem Exception-Handler mittels des Befehls RTE (siehe unten).

7.3 ARBEITUNG VON EXCEPTIONS

Bei allen auftretenden Exceptions führt der Prozessor fast die gleichen Operationen durch. Zunächst werden in allen Fällen die Inhalte des Programmzählers und des Statusregisters gerettet, so daß das unterbrochene Programm später wieder aufgenommen werden kann. Diese Rettung erfolgt unabhängig vom momentanen Modus auf den **Supervisor-Stack**, d.h. auf den durch SSP adressierten Stack. Die Abspeicherung geschieht in der folgenden Reihenfolge



Bei Adreß- und Busfehlern werden noch weitere Informationen auf den Stack gerettet (siehe unten). In den meisten Fällen zeigt der Programmzähler auf den Befehl, der bei normaler Abarbeitung als nächster auszuführen gewesen wäre. 68010 und 68020 legen ein wieteres Wort auf dem Stack ab, das die Nummer des Exceptionvektors enthält. Dadurch wird die Benutzung desselben Codes für verschiedene Exception-Handler erleichert.

Nachdem der alte Inhalt gerettet wurde, wird das Statusregister auf einen bestimmten, fest vorgegebenen Wert gesetzt. Das Supervisor-Bit ist dann gesetzt, so daß das angesprungene Unterprogramm im Supervisormodus ablaufen kann. Das Trace-Bit wird gelöscht, so daß die Ausnahmebedingungen im normalen Ablauf behandelt werden können, auch wenn das Hauptprogramm getract wird (siehe unten). Die drei Bits der Interruptmaske werden nur dann beeinflußt, wenn ein Reset oder ein Interrupt auftritt. Bei Reset wird diese Maske auf den Wert 111 gesetzt, wogegen ein Interrupt dort eine Angabe über die Priorität dieses Interrupts ablegt.

7.4 EXCEPTION-HANDLER

Ein Programm, das die möglichen Exceptions in eigenen Unterprogrammen behandeln will, muß zunächst die Adressen dieser Unterprogramme in der Vektortabelle ablegen. Das Grundgerüst eines solchen Unterprogramms sieht wie folgt aus:

```
ENTRYPT  MOVEM.L  Dp-Dq/Ar-As,-(SP)  Alle benötigten Register
*                                    retten
         :
         :                           Exception behandeln
         :
         MOVEM.L  (SP)+,Dp-Dq/Ar-As  Register wiederherstellen
         RTE                         Rücksprung ins unterbrochene
*                                    Programm
```

Eine Exception kann zu einem beliebigen Zeitpunkt eintreten, auch mitten in der Ausführung beliebiger Befehle. Um später das unterbrochene Programm richtig wiederaufnehmen zu können, ist es wichtig, das Statusregisters und alle Daten- und Adreßregister wieder in ihren ursprünglichen Zustand zu versetzen. Das Statusregister wird automatisch gerettet; aber es ist Sache des angesprungenen Unterprogramms, sich um die anderen Register zu kümmern. Am einfachsten geschieht dies durch Retten aller in dem Unterprogramm verwendeten Register auf dem Stack.

Der Befehl RTE bewirkt die Rückkehr in das unterbrochene Programm. Der alte Programmzähler und der Programmstatus müssen dazu in der oben beschriebenen Reihenfolge als oberste Werte auf dem Stack liegen. Bei Ausführung von RTE werden diese beiden Register vom Stack geladen, wodurch das unterbrochene Programm seine Arbeit wieder aufnimmt. RTE hat also etwa die Auswirkung der Befehlsfolge:

```
MOVE.W   (SP)+,SR
RTS
```

RTE läßt sich jedoch nicht durch diese beiden Befehle ersetzen, da RTE die Rücksprungadresse immer vom **Supervisor-Stack** und nicht wie RTS in Abhängigkeit vom jeweiligen Modus von einem der beiden Stacks nimmt. Da der Befehl MOVE.W mit Zieloperand SR das Supervisor-Bit ändert, würde RTS anschließend evtl. den falschen Stack ansprechen.

Die Verwendung von RTE ist im Benutzermodus nicht erlaubt, da durch diesen Befehl das Statusregister modifizierbar ist und somit den ungehemmten Zugang zum Supervisormodus ermöglicht.

Ein RTE ähnlicher, aber nichtprivilegierter Befehl ist RTR. Auch er lädt Programmzähler und Statusregister vom Stack, von letzterem jedoch nur das Benutzer-Byte (also die Condition-Codes). RTR holt dafür ein ganzes Wort vom Stack, verwendet aber nur dessen unteres Byte. RTR wirkt also wie

```
        MOVE.W    (SP)+,CCR
        RTS
```

RTR ist eine Alternative zu RTS für den Rücksprung aus einem Unterprogramm, das als erste Operation bei Einsprung den Inhalt des Statusregisters auf dem Stack rettet. Ein solches Unterprogramm hat dann die folgende Struktur:

```
UPRNAME MOVE.W    CCR,-(SP)   Condition-Codes retten
        :
        :
        RTS                   Condition-Codes wiederherstellen
*                             und Rücksprung
```

RTE, RTR und RTS lassen sich auch als Sprungbefehle benutzen. Um an eine Adresse zu springen, deren Wert sich erst im Laufe der Berechnung ergeben hat, muß dieser Wert normalerweise in einem Adreßregister vorliegen. Wenn wir jedoch bereits sämtliche Register mit irgendwelchen für das Unterprogramm wichtigen Werten belegt haben, können wir auch die Sprungadresse auf dem Stack ablegen und anschließend RTS ausführen.

```
        MOVE.L    zieladresse,-(SP)  Sprungziel auf den Stack
        :
        :                     Register laden
        :
        RTS                   Sprung
```

RTR und RTE können so benutzt werden, um beim Sprung die Condition-Codes oder (im Supervisormodus) das ganze Statusregister zu verändern.

7.5 INTERRUPTS

Interrupts (Unterbrechungen) sind die Ereignisse, mit denen externe Einheiten die Aufmerksamkeit des Prozessors erregen. Ein Gerät versucht, einen Interrupt zu verursachen, indem es dem Prozessor eine Prioritätsangabe zwischen 1 und 7 übermittelt. Der Interrupt wird nur dann akzeptiert, wenn die momentan im Prozessor vorliegende Prioritätsangabe (die Interruptmaske im Statusregister) kleiner als die Prioritätsangabe der Anforderung ist oder aber diese Anforderung Priorität 7 hat. Priorität 7 wirkt wie ein **nichtmaskierbarer Interrupt** (**NMI**, non-maskable interrupt) bei anderen Computern. Die den Interrupt anfordernde externe Logik muß dem Prozessor zusätzlich mitteilen, ob sie eine Vektornummer angeben will oder das sogenannte **'Auto-Vectoring'** verwendet werden soll. Dabei benutzt der Prozessor in Abhängigkeit von der Prioritätsangabe eine Vektornummer zwischen 25 und 31. Wenn viele Einheiten dieselbe Prioritätsangabe verwenden, wird normalerweise eine Vektornummer verwendet.

Die Abarbeitung eines Interrupts entspricht der jeder anderen Exception: Programmzähler und Statusregister werden auf den Stack gerettet, und das Statusregister

erhält einen fest vordefinierten Wert. Die Prioritätsangabe im Statusregister wird auf den von dem Gerät angegebenen Wert gesetzt. Dies geschieht, um während der Behandlung des Interrupts weitere Interrupts der gleichen oder einer geringeren Priorität zu unterbinden, solche mit einer höheren Priorität dagegen zuzulassen. Einige periphere Geräte benötigen eine sofortige Reaktion des Prozessors auf ihren Interrupt, da sonst die Gefahr des Datenverlusts besteht, wogegen andere Geräte beliebig lange darauf warten können, daß sie vom Prozessor bedient werden. Der Entwickler eines Rechnersystems muß die Interrupt-Prioritäten so verteilen, daß der Interrupt durch ein Gerät mindestens so schnell wie nötig behandelt wird. Interrupt-Anforderungen der gleichen oder einer niedrigeren Priorität werden ignoriert, um den Aufbau der Interrupt-Handler zu vereinfachen. Falls weitere Interrupts der gleichen Stufe eintreffen, könnte sonst leicht der Fall eintreten, daß ein Unterprogramm unterbrochen und selbst wieder aufgerufen wird. Dies ergäbe ein gewaltiges Durcheinander, wenn dieses Unterprogramm die vermeintlich exklusive Kontrolle über gewisse Speicherbereiche oder periphere Geräte besitzt.

7.6 EXTERNER RESET

Durch einen Schaltkreis außerhalb des Prozessors kann ein externer Reset verursacht werden. Diese Exception wird benutzt, um den Prozessor zu initialisieren oder ihn »wieder auf die Füße zu stellen«, wenn ein anders nicht behebbarer Systemabsturz vorliegt. Jegliche zum Zeitpunkt eines Resets im Prozessor vorhandene Information geht dabei unwiderruflich verloren.

Der Reset unterscheidet sich in zwei Punkten von allen anderen Exceptions: Zunächst wird nichts auf dem Stack gerettet, da der Stackzeiger unter Umständen auf keine gültige Adresse zeigt. Des weiteren ist der Exceptionvektor in der Tabelle nicht vier, sondern acht Byte lang. Er enthält den initialen Wert des Stackzeigers für den Supervisor und einen neuen Programmzähler.

7.7 ILLEGALE UND NICHT IMPLEMENTIERTE BEFEHLE

Versucht der 68000 ein Wort auszuführen, daß keinen gültigen Befehl enthält, so verursacht er einen Trap, wobei der gerettete Programmzähler auf den falschen Befehl zeigt. Einer von drei verschiedenen Exceptionvektoren wird zum Anspringen eines Unterprogramms verwendet.

Enthalten die obersten vier Bits des monierten Wortes das Bitmuster 1010 oder 1111, so wird dieser Befehl nicht als illegal, sondern als **nicht implementiert** behandelt. Diese Befehle können in zukünftigen Modellen des 68000 eine Bedeutung erhalten oder zur Ansteuerung sogenannter **Co-Prozessoren** gedacht sein. Nichtimplementierte Befehle verursachen in Abhängigkeit von den genannten vier Bits den Aufruf eines von

zwei verschiedenen Unterprogrammen. Dadurch können unimplementierte Befehle softwaremäßig emuliert werden. Bei gegebener Spezifikation eines Befehls kann ein Unterprogramm zur Behandlung dieses Traps geschrieben werden, das die gleiche Auswirkung wie die Ausführung dieses Befehls hat, jedoch wesentlich langsamer ist. Somit ist es möglich, Programme mit unimplementierten Befehlen zu schreiben, die auf späteren Modellen des 68000 ohne Änderung ablaufen können.

Illegale Befehle, die nicht in eine dieser beiden Gruppen fallen, verursachen einen eigenen Trap. Das Auftreten dieser Exception ist normalerweise ein Symptom für einen wilden Sprung (meistens durch indirekte Adressierung mit einem nicht initialisierten Adreßregister) oder aber einen vergessenen Abschluß des Programmes (Rücksprung ins Betriebssystem oder zum Monitor), so daß das Programm in unbekannte Regionen oder Datenbereiche gerät.

7.8 BEFEHLE, DIE EINEN TRAP VERURSACHEN KÖNNEN

Einige Befehle können im Zuge ihrer normalen Abarbeitung einen Trap verursachen, weil dies entweder ihre Aufgabe ist, oder aber, weil bei ihrer Ausführung eine Unregelmäßigkeit auftritt.

Der Befehl TRAP führt generell eine Exception herbei. Sein Operand ist eine Zahl zwischen 0 und 15, und einer von 16 Exceptionvektoren wird zur Adressierung eines Unterprogramms verwendet, so daß es eigentlich eine Familie von 16 TRAP-Befehlen gibt. Er kann zu vielen Zwecken verwendet werden, meist wird er jedoch zum Aufruf einer Betriebssystem- oder Monitorroutine benutzt. Wie wir bereits früher erwähnt hatten, muß in einem geschützten System das Betriebssytem im Supervisormodus ablaufen, um die Aktionen der Benutzerprogramme einschränken zu können. Der TRAP-Befehl ermöglicht den Benutzerprogrammen, Routinen des Betriebssystems aufzurufen und als Teil des Aufrufs den Prozessor in den Supervisormodus zu versetzen. Der Grund des Aufrufs kann dem Betriebssystem durch den Operanden von TRAP oder durch explizite Übergabe eines Wertes in einem Register bekannt gemacht werden.

TRAP dient auch dazu, in einem Debugging-Programm **Breakpoints** zu setzen. Da dieser Befehl nur zwei Byte lang ist, kann das erste Wort eines jeden Befehls durch ihn ersetzt werden. Erreicht das Programm diese Stelle, so tritt eine Exception ein, wodurch die Kontrolle wieder an den Debugger zurückgeht, der dann eine adäquate Meldung ausgibt. Der im nächsten Kapitel vorgestellte Monitor verwendet derartige Breakpoints.

Zwei weitere Befehle verursachen einen Trap, falls eine bestimmmte Bedingung gilt. Diese beiden Befehle sind als billige (was Zeit und Platz betrifft) Fehlertests für ein laufendes Programm gedacht; sie könnten daher von einem Compiler hinter alle kritischen Befehle des eigentlichen Programms plaziert werden. TRAPV verursacht eine

Exception, falls das V-Bit im Statusregister gesetzt ist. Fügt man hinter jedem arithmetischen Befehl ein TRAPV ein, so erhält man bei jedem Überlauf einen Trap. Derartige Programmstücke sehen etwa folgendermaßen aus:

```
ADD.L     (A1),D4
TRAPV
ASL.L     #2,D4
TRAPV
:
:
```

CHK ist der andere dieser Befehle, der zur Verifikation eines Index, d.h. zur Kontrolle, ob der Index noch innerhalb eines erlaubtes Bereiches liegt, vor einem Arrayzugriff gedacht ist. CHK vergleicht den durch den Quelloperanden gegebenen Wert mit den unteren 16 Bits des Datenregisters, das als Zieloperand verwendet wird. Ist der Inhalt des Datenregisters negativ oder größer als der Quelloperand, so tritt ein Trap auf.

Im folgenden Beispiel enthält A1 die Anfangsadresse eines Arrays, D1 den Offset des Elements, auf das wir zugreifen wollen, und das durch A2 adressierte Wort enthält den größten möglichen Index in diesem Array. Dann kann CHK wie folgt eingesetzt werden:

```
CHK       (A2),D1          Test, ob der Index gültig ist
MOVE.B    WERT,0(A1,D1.W)     Schreibender Zugriff
```

Die Divisionsbefehle DIVU und DIVS können auch eine Exception verursachen, wenn der Versuch der Division durch Null unternommen wird.

Bei allen oben aufgeführten Befehlen zeigt der gerettete Programmzähler auf den nächsten Befehl.

7.9 PRIVILEGVERLETZUNG

Wie wir bereits öfters erwähnten, können einige Befehle nur im Supervisormodus ausgeführt werden. Dies sind all die Befehle, welche die absolute Kontrolle über den Computer ermöglichen (und somit dem Betriebssytem vorbehalten sein müssen, damit es nicht von einem anderen Programm einfach rausgeschmissen wird) oder die periphere Geräte zurücksetzen können. Im einzelnen sind dies die Befehle:

```
RESET
STOP      #XXXX
RTE
MOVE.W    <OA>,SR
ANDI.W    #WORT,SR
ORI.W     #WORT,SR
EORI.W    #WORT,SR
MOVE.L    USP,An
MOVE.L    An,USP
```

Der Befehl RESET beeinflußt den Reset-Ausgang des Prozessors, wodurch alle angeschlossenen Geräte in einen Standardzustand gelangen. Normalerweise ist er einer der ersten Befehle, die beim Start eines Betriebssytems oder Monitors ausgeführt werden. Im normalen Programmablauf wird er nicht mehr verwendet. Dieser Befehl steht in keinerlei direktem Zusammenhang zu der durch einen externen Reset verursachten Exception, sollte aber in dem Exception-Handler verwendet werden, um sicherzustellen, daß alle angeschlossenen Geräte sich in einem korrekten Zustand befinden.

Der Befehl STOP versetzt den Prozessor in den sogenannten **gestoppten** Zustand. Dieser darf nicht mit dem **angehaltenen** Zustand verwechselt werden, in den der Prozessor nach einem nicht behebbaren Fehler gelangt. Der Prozessor stoppt die Befehlsausführung solange, bis ein Interrupt oder ein Reset eintritt. Der Operand von STOP ist ein 16-Bit-Wert, der im Statusregister abgelegt wird. STOP kann so vor dem Anhalten des Prozessors eine Prioritätsangabe für Interrupts setzen.

Dieser Befehl ist nur dann sinnvoll verwendbar, wenn das um den 68000 aufgebaute Computersystem Geräte besitzt, die zu **direktem Speicherzugriff (DMA**, direct memory access) fähig sind. Derartige Geräte können ohne Hilfe des Prozessors den Speicher direkt lesen oder schreiben. Plattenlaufwerke sind normalerweise unter Verwendung von DMA angeschlossen, damit große Datenmengen zwischen Speicher und Platte transportiert werden können, ohne das laufende Programm zu beeinträchtigen. Dieses wird nur bei Beendigung des Transfers unterbrochen, um es über die abgeschlossene Übertragung zu informieren. Da eine DMA-Einheit häufig gleichzeitig mit dem Prozessor auf den Speicher zugreifen will, entscheiden einige Schaltkreise, wer dies darf und wer warten muß. Wenn ein Programm einen DMA-Transfer veranlaßt und anschließend bis zum Abschluß dieses Transfers unbeschäftigt ist, könnte es einfach in eine Schleife gehen und auf den Interrupt warten. Dadurch werden jedoch unnötige Speicherzugriffe verursacht (um die Befehle zur Schleifenbildung zu lesen) und der DMA-Transfer verlangsamt. Wenn der 68000 tatsächlich nichts zu tun hat, versetzt er sich daher besser mit dem Befehl STOP in den gestoppten Zustand.

Der Operand von STOP muß eine Eins an der Stelle besitzen, der im Statusregister das Supervisor-Flag entspricht. Anderenfalls tritt eine Privilegverletzung auf, auch wenn sich der Prozessor im Supervisormodus befindet. (Dies ist übrigens die einzige Möglichkeit, im Supervisormodus eine Privilegverletzung zu erzeugen). Ein typisches Beispiel für diesen Befehl ist daher

```
STOP    #$2000
```

der die Interrupt-Maske auf Null setzt und somit jeden Interrupt zuläßt.

Die MOVE-Befehle mit dem Operanden USP müßten eigentlich nicht privilegiert sein. Sie sind jedoch nur zur Verwendung im Supervisormodus gedacht, da der Stackzeiger im Anwendermodus ohne weiteres als SP (oder A7) adressierbar ist. Da der

Versuch, einen dieser Befehle im Anwendermodus auszuführen, vermutlich auf einem Programmfehler beruht, auf den der Benutzer aufmerksam gemacht werden sollte, wurden sie zu den privilegierten Befehlen genommen.

Bei Auftreten einer Privilegverletzung zeigt der gerettete Programmzähler auf den monierten Befehl.

7.10 TRACEN

Der 68000 kann veranlaßt werden, nach der Ausführung eines jeden Befehls eine Exception zu erzeugen. Dies geschieht, wenn das **Trace-Bit** im Statusregister gesetzt ist. Der gerettete Programmzähler zeigt dann auf den nächsten auszuführenden Befehl.

Dieser Modus wird hauptsächlich bei der Fehlersuche verwendet, da so ein auszutestendes Programm Schritt für Schritt ausgeführt werden kann, wobei die Kontrolle nach jedem Befehl wieder an den Debugger geht. Dies ist ein mächtiges Mittel, um in einem Programm eine fehlerhafte Stelle zu lokalisieren. Der im nächsten Kapitel vorgestellte Monitor verwendet den Tracemodus in diesem Sinne.

Der **Tracemodus** vereinfacht auch die Behandlung der Breakpoints. Wir sahen bereits, wie der Befehl TRAP das erste Wort eines Befehles ersetzen kann, um dort einen Breakpoint zu setzen. Bei Erreichen dieses Befehls geht die Kontrolle wieder an den Debugger, und wir können Register, Speicherstellen und ähnliches untersuchen und verändern. Es bleibt jedoch das Problem, das Programm hinter dem Breakpoint wiederaufzunehmen. Normalerweise soll der Breakpoint aktiviert bleiben, da das Programm wieder an diese Stelle gelangen kann. Um im Programm fortzufahren, müssen wir zunächst den Befehl ausführen, der durch den TRAP ersetzt wurde. Dazu müssen wir den Befehl wiederherstellen, ihn ausführen und anschließend wieder durch TRAP ersetzen, bevor das weitere Programm abgearbeitet wird. Besteht die Möglichkeit zu tracen, ist dies recht einfach zu bewerkstelligen: Der Befehl wird wiedererstellt und mit eingeschaltetem Tracemodus ausgeführt. Nach der Ausführung geht die Kontrolle wieder an den Debugger zurück, und dieser kann den Befehl wieder durch TRAP ersetzen, das Trace-Bit im Statusregister löschen und den normalen Programmablauf wiederaufnehmen.

7.11 BUS- UND ADRESSFEHLER

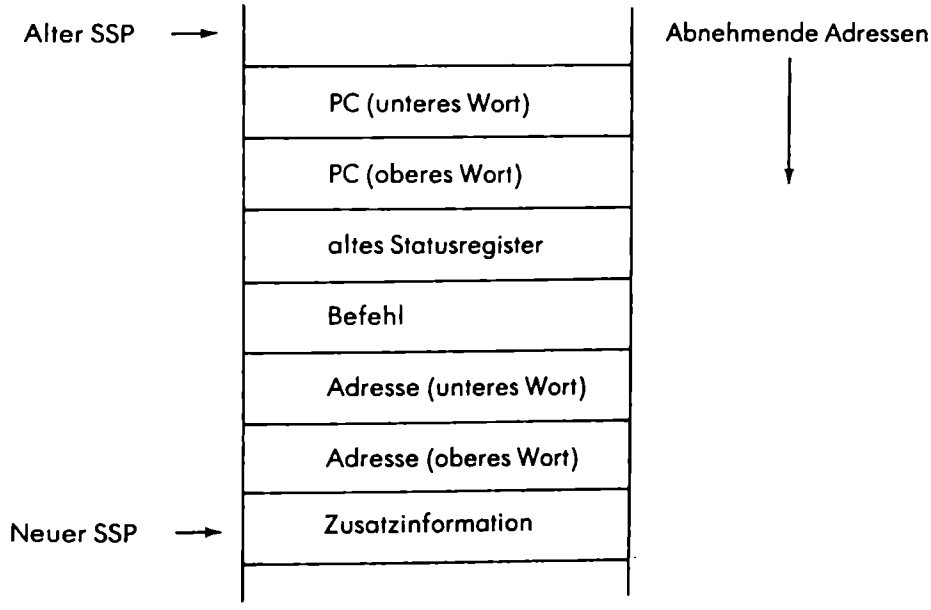
Ein Busfehler tritt auf, wenn eine weder als Speicher noch als (Kontroll- oder Daten-) Register existente Adresse angesprochen wird. Normalerweise tritt er auf, wenn versucht wird, außerhalb des vorhandenen Speichers zu adressieren. Dieser Fehler kann nur dadurch entdeckt werden, daß auf das Ansprechen einer Adresse keine Antwort von irgendeiner Einheit auf dem Prozesorbus kommt. Dieser Test wird von Schaltkreisen außerhalb des Prozessor vorgenommen, so daß der Entwickler selbst entscheiden kann,

wie lang eine noch akzeptable Wartezeit sein darf. soll. Wäre diese Zeitkonstante im Prozessor fest eingebaut, so wäre die Verwendung langsamerer Einheiten unmöglich.

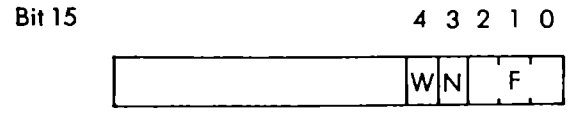
Als Adreßfehler wird der Versuch bezeichnet, auf ein Wort oder Vollwort an einer ungeraden Adresse zugreifen (lesend oder schreibend) zu wollen. Dieser Fehler ähnelt dem Busfehler, wird aber vom Prozessor selbst entdeckt und benutzt einen anderen Exceptionvektor.

Meist ist es sehr schwierig, die genaue Ursache eines Bus- oder Adreßfehlers zu lokalisieren. Sie können durch ungültige Operandenadressen im laufenden Befehl, aber auch beim Versuch auftreten, mittels 'pre-fetch' ein Wort zu holen. Während die meisten Exceptions zwischen zwei Befehlen abgearbeitet oder durch einen Befehl verursacht werden, können diese Fehler zu einem beliebigen Zeitpunkt während der Abarbeitung eines Befehls auftreten. Der gerettete Programmzähler zeigt daher irgendwo in die Gegend des monierten Befehls; es läßt sich aber nicht sagen, wohin genau.

Zur Vereinfachung der Fehlersuche rettet der Prozessor bei diesen beiden Fehlern mehr Informationen als üblich auf dem Stack. Insgesamt werden die folgenden sieben Worte abgelegt:



Das mit 'Befehl' bezeichnete Wort enthält das erste Wort des Befehls, während dessen Abarbeitung der Bus- oder Adreßfehler auftrat. Dadurch kann der Exception-Handler vom geretteten Programmzähler an rückwärts suchen, um den Befehl im Speicher zu finden. Das Feld 'Adresse' enthält die Adresse, auf die versucht wurde zuzugreifen. Das letzte Wort enthält weitere Informationen über den abgebrochenen Buszyklus; es hat das folgende Format:



Das W-Bit gibt an, ob es sich um einen schreibenden (gesetzt) oder lesenden (gelöscht) Zugriffsversuch handelte. Das N-Bit ist gesetzt, wenn der 68000 weder mit der Abarbeitung eines Befehls noch mit der unmittelbaren Behandlung einer Exception beschäftigt war, als der Fehler auftrat; der Prozessor könnte also beispielsweise im gestoppten Modus gewesen sein. Das F-Feld enthält den Zustand der bereits früher erwähnten Ausgabeleitungen zur Speicherauswahl im Augenblick des Fehlers. Diese Leitungen klassifizieren den Zugriff im Supervisor- oder Benutzermodus, auf Programm- oder Datenspeicher, und können so den Adreßraum in vier Bereiche aufspalten.

Enthält der Exceptionvektor für Busfehler, Adreßfehler oder Reset eine illegale oder ungerade Adresse, so tritt ein Bus- oder Adreßfehler direkt beim Versuch der Behandlung der Exception auf. Dies wird als 'doppelter' Busfehler bezeichnet und als nicht behebbarer Fehler behandelt. Der Prozessor gibt auf und hält sich selbst an, um nicht irgendwelche wichtigen Speicherinhalte zu zerstören. Das System läßt sich in diesem Zustand nur durch ein externes Reset-Signal neustarten.

Im allgemeinen kann der 68000 einen Befehl, der einen Busfehler verursachte, nicht fortführen, da der Fehler während der internen Abarbeitung auftrat. 68010 und 68020 speichern wesentlich mehr Informationen auf dem Stack, wodurch dieser Befehl nach Behandlung des Fehlers doch wieder aufgenommen werden kann.

7.12 MEHRFACHE EXCEPTIONS

Mitunter treten mehrere Exceptions sehr schnell hintereinander ein, und dann kann die Reihenfolge ihrer Abarbeitung bedeutend werden. Die verschiedenen Exceptions können in Abhängigkeit von ihrem Behandlungszeitpunkt in drei Gruppen eingeteilt werden.

Gruppe 0: Reset, Busfehler, Adreßfehler. Die Ausführung des augenblicklichen Befehls wird abgebrochen.

Gruppe 1: Trace, Unterbrechungen, Privilegverletzungen, illegale Befehle
Der augenblickliche Befehl wird zu Ende ausgeführt; danach tritt die Exception ein. Privilegverletzungen sowie Trapsaufgrund illegaler Befehle treten unmittelbar vor dem Ausführen des unerlaubten Befehls auf.

Gruppe 2: TRAP, TRAPV, CHK, Division durch Null
Die Exception tritt als Teil der normalen Programmausführung auf.

Es gibt eine Prioritätenregelung für Exceptions, die festlegt, was geschieht, wenn mehrere Exceptions gleichzeitig anstehen. Die Exception mit der höchsten Priorität wird als erste behandelt, dann erst die anderen. Diese Reihenfolge lautet (in absteigender Folge):

Reset
Busfehler
Adreßfehler
Trace
Interrupt
Illegaler Befehl, Privilegverletzung
TRAP, TRAPV, CHK, Division durch Null

Die in den beiden letzten Zeilen stehenden Urachen können nicht gemeinsam auftreten, so daß ihnen eine gemeinsame Priorität zugeordnet werden kann.

Beim Auftreten eines Resets, Bus- oder Adreßfehlers wird das evtl. gesetzte Trace-Bit nicht beachtet. Liegt nach einem getraceten Befehl eine Interrupt-Anforderung vor, so wird zunächst der Trace behandelt. Verursachte der momentane Befehl eine Exception, so wird diese vor dem Trace abgearbeitet.

7.13 EIN UNTERPROGRAMM ZUR ERMITTLUNG DER SPEICHERGRÖSSE

Wir können die Exception »Busfehler« in einem Unterprogramm zur Feststellung der Speichergröße unseres Computers einsetzen. Ein derartiges Unterprogramm wird normalerweise vom Betriebssystem ausgeführt, so daß das gleiche Betriebssystem auf Computern mit verschiedener Speichergröße ablaufen kann, ohne jemals etwas von dem vorhandenen Speicher ungenutzt zu lassen.

Dieses Unterprogramm geht durch den Speicher und versucht jedes Byte anzusprechen. Höchstwahrscheinlich wird einmal ein Byte zuviel angesprochen, das gar nicht existiert, wodurch ein Busfehler verursacht wird. Vor dem Eintreten in diese Hauptschleife wird der Sprungvektor für das Unterprogramm zur Behandlung des Busfehlers auf einen Befehl in diesem Unterprogramm gesetzt (beim Label SSF__BERR). Da wir nichts von der beim Auftreten eines Busfehlers geretteten Information benötigen, können wir uns dort darauf beschränken, den Stackzeiger wieder auf seinen alten Wert zu setzen.

Es kann vorkommen, daß aufgrund des Aufbaus der Adreßdekodierungslogik die Logik zum Erkennen eines Busfehlers keine Versuche entdeckt, Speicher oberhalb der höchsten existierenden Adresse anzusprechen. Daher prüft unser Unterprogramm jedes einzelne Byte darauf, ob es auch wirklich als Speicherstelle fungiert. Wir verwenden dazu ein aus Einsen und Nullen bestehendes Bitmuster, da sich nicht existierender Speicher gewöhnlich als mit lauter Nullen oder lauter Einsen gefüllt gibt. Der Originalinhalt jeder Speicherstelle wird in einem Register gerettet und nach dem Test wieder abgespeichert, um den Speicherinhalt durch den Test nicht zu verändern. Von diesem Test müssen wir den Bereich ausnehmen, in dem unser Unterprogramm liegt, da wir sonst an einer Stelle ein Byte des nächsten auszuführenden Befehls ändern würden, was sicherlich zu unerwünschten Ergebnissen führen würde. Das Unterprogramm beginnt daher mit dem Test hinter seinem letzten Byte.

```
* Dieses Unterprogramm setzt A0 auf die Größe des vorhandenen
* Speichers (in Byte)
*
* Der Test beginnt unmittelbar hinter dem Unterprogramm und geht
* von einem zusammenhängenden Speicher aus

MEMPAT   EQU      $AA             Bitmuster für den Speichertest
I_BERR   EQU      $8              Adresse des Exceptionvektors für
*                                 einen Busfehler

* Installieren des Exceptionvektors

         MOVE.L   I_BERR,D0       Alten Vektor in D0 retten
         MOVEA.L  SSP,A           Alten Stackzeiger retten
         LEA.I    SSF_BERR,A0     Adresse für Busfehler
         MOVE.L   A0,I_PBERR      Installieren des Vektors

* Anfang der Suche hinter diesem Unterprogramm

         LEA      SSF_END,A0      Anfangsadresse

* Hauptschleife

SSFLOOP  MOVE.B   (A0),D1         Byte retten (verursacht evtl.
*                                 einen Busfehler)
         MOVE.B   #MEMPAT,(A0)    Bitmuster ablegen
         CMPI.B   #MEMPAT,(A0)    Ist bei A0 Speicher ?
         BNE.S    SSF_FOUND       Nein → Speichergrenze gefunden
*
         MOVE.B   D1,(A0)+        Speicherinhalt wiederherstellen,
*                                 A0 weitersetzen
         BRA.S    SSFLOOP         Nächstes Byte

* Bei Busfehler erfolgt hier der Einsprung zur Behandlung

SSF_BERR MOVEA.L  A6,SP           Stackzeiger zurücksetzen

* Speichergröße ermittelt (steht in A0)

SSF_FOUND MOVE.L  D0,I_BERR       Vektor wiederherstellen
SSF_END
```

8

Ein komplettes Programm: Ein Monitor

In diesem Kapitel wollen wir Ihnen ein komplettes Programm vorstellen. Es handelt sich dabei um einen kleinen Monitor mit begrenzten Möglichkeiten, Assemblerprogramme ablaufen zu lassen und zu debuggen. Ein solcher Monitor liegt normalerweise im ROM des Computers vor und besitzt einige Befehle mehr als der hier vorgestellte. Dennoch zeigt dieses Beispiel anschaulich die Verwendung der Interrupt- und Trapvektoren, des Supervisor- und des Benutzermodus des 68000.

Der Monitor soll an einer beliebigen Speicherstelle ablaufen können; daher wurde einiger Wert darauf gelegt, relokatiblen Code zu erzeugen. Der Monitor selbst lauft im Supervisormodus, wogegen ein Anwenderprogramm nur im Benutzermodus laufen kann.

Es sind Befehle vorgesehen, den Inhalt des Registersatzes des Benutzers oder Speicherstellen zu untersuchen und zu verändern, Breakpoints zu setzen und zu löschen, und um ein Programm im normalen oder im Tracemodus zu starten.

Durch Eingabe des R-Befehls werden die augenblicklichen Inhalte aller Benutzer-Register ausgegeben. Alternativ kann mit Hilfe des A- oder D-Befehls der Inhalt eines einzelnen Adreß- oder Datenregister ausgegeben werden. Programmzähler und Statusregister können durch Eingabe des P- bzw. S-Befehls ausgegeben werden.

Der Inhalt eines beliebigen Registers kann verändert werden, indem nach Eingabe des A-, D-, P- oder S-Befehls ein hexadezimaler Wert angegeben wird. Zur Behandlung des Speichers muß dieser zunächst an irgendeiner Stelle »geöffnet« werden, d.h. die Anfangsadresse des Speicherbereiches wird dem Monitor mitgeteilt. Dies geschieht durch Eingabe des M-Befehls, worauf der Monitor zunächst mit der Ausgabe des an der angegebenen Adresse liegenden Bytes reagiert. Nachdem eine Speicherstelle erstmalig geöffnet wurde, wird der Befehlssatz zur Behandlung des Speichers wirksam. Ursprünglich wurde die Speicherstelle als Byte-Wert geöffnet, aber das an dieser Stelle liegende Wort oder Vollwort kann durch Eingabe von W bzw. L spezifiziert werden. S reduziert das Format wieder auf ein einzelnes Byte. Ungerade adressierte Speicherstellen können natürlich nicht als Wort oder Vollwort geöffnet werden.

Monitorbefehle	
Ar Ar n	Ausgabe des Inhalts von Benutzer-Adreßregister r Setzt Benutzer-Adreßregister r auf n
B Bi Bi n	Ausgabe der gesetzten Breakpoints Breakpoint i löschen Breakpoint i an der Adresse n setzen
C	Fortfahren nach einem Breakpoint
Dr Dr n	Ausgabe des Inhalts von Benutzer-Datenregister r Setzt Benutzer-Datenregister r auf n
G G n	Start des Benutzer-Programms beim Benutzer-Programmzähler Start des Benutzer-Programms an der Adresse n
M n	Öffnen der Speicherstelle n (Nun sind die Speicherbefehle gültig)
P P n	Ausgabe des Inhalts des Benutzer-Programmzählers Setzt Benutzer-Programmzähler auf n
R	Ausgabe der Inhalte aller Benutzer-Register
S S n	Ausgabe des Inhalts des Benutzer-Statusregisters Setzt Benutzer-Statusregister auf n
T T n	Trace des Benutzer-Programms ab Benutzer-Programmzähler Trace des Benutzer-Programms ab Adresse n

Wurde eine Speicherstelle geöffnet, so kann ihr durch Eingabe einer Hexadezimalzahl ein neuer Wert verliehen werden. Je nach der eingestellten Größe werden dadurch ein, zwei oder vier Bytet. Das »Pfeil-nach-oben«-Zeichen (^) öffnet die vorhergehende Speicherstelle in der eingestellten Größe, Return die nachfolgende. Das Gleichheitszeichen (=) kann benutzt werden, um den Inhalt der momentan geöffneten Speicherstelle erneut auszugeben. Dies ist sinnvoll, falls es sich dabei nicht um eine echte Speicherstelle, sondern vielmehr um ein Datenregister eines externen Geräts handelt. Die Unterebene der Speicherbefehle kann durch Eingabe eines Punktes (.) beendet werden. Es erfolgt dann die Rückkehr zum normalen Monitor.

Bei diesem Monitor muß ein Benutzer-Programm mit Hilfe des M-Befehls Byte für Byte in den Speicher eingegeben werden; andere Monitore bieten die Möglichkeit, ein Programm von einer externen Einheit (Platte, Band oder ein anderer Computer) zu laden.

Speicherbefehle	
return	Geht zur nächsten Speicherstelle
^	Geht zur vorherigen Speicherstelle
=	Gibt den Inhalt der geöffneten Speicherstelle aus
.	Verläßt die Unterebene der Speicherbehandlung
n	Setzt die geöffnete Speicherstelle auf den Wert n
L	Einstellen des Vollwortformats
W	Einstellen des Wortformats
B	Einstellen des Byteformats

Liegt ein Programm erst einmal im Speicher, so kann es mit Hilfe des G-Befehls ausgeführt werden. Dieser versetzt den Prozessor in den Benutzermodus und verursacht einen Sprung zu der durch den Benutzer-Programmzähler spezifizierten Adresse. Für diesen kann nach dem 'G' bei Bedarf auch ein neuer Wert angegeben werden, der dann vor dem Sprung in den Zähler geladen wird.

Zum Debuggen eines Programms kann auch der T-Befehl zum Start der Ausführung verwendet werden. In diesem Fall wird ein Befehl ausgeführt und die Kontrolle sofort wieder an den Monitor übergeben. Dieser gibt dann die Inhalte der Register aus, und ein beliebiger anderer Monitorbefehl kann gegeben werden. Bei Eingabe eines einfachen Returns an dieser Stelle wird der nächste Befehl ausgeführt. Dies ermöglicht es, ein Programm Schritt für Schritt abzuarbeiten.

Der Monitor ermöglicht auch das Setzen von Breakpoints im Anwenderprogramm. Der B-Befehl alleine gibt alle momentan gesetzten Breakpoints aus, 'Bi' löscht den Breakpoint i, und 'Bi' gefolgt von einer Hexadezimalzahl setzt einen Breakpoint an der angegebenen Adresse. Bei Erreichen des Breakpoints im normalen Programmablauf geht die Kontrolle zurück an den Monitor, wonach das Programm durch Eingabe des C-Befehls wieder aufgenommen werden kann.

Sämtliche Exceptions werden behandelt. TRAP #15 wird benutzt, um dem Monitor das Ende eines Benutzer-Programms zu signalisieren, und TRAP #14 wird zur Realisierung der Breakpoints verwendet. Alle anderen Exceptions (außer TRACE, der zum Tracen des Benutzer-Programmes benötigt wird) verursachen die Ausgabe einer entsprechenden Meldung und das Terminieren des Programms, wobei die Registerinhalte gerettet werden.

Von den Interruptvektoren wird nur der Auto-Vektor der Stufe zwei verwendet. Wir gehen hier davon aus, daß ein Interrupt des ACIAs über diesen Vektor behandelt wird. Die Adresse des ACIAs liegt als Vollwortkonstante im Programm vor, so daß diese bei Bedarf geändert werden kann. Die vom Monitor für Stack und Datenbereiche vorgesehenen Speicherabschnitte sind durch zwei weitere Vollwortkonstanten im Programm definiert und können somit ebenfalls leicht geändert werden.

8.1 KONSTANTENDEFINITIONEN UND EXCEPTION-HANDLER

Zunächst müssen wir einige benötigte Konstanten definieren und das Layout des Speichers definieren. In diesem ersten Abschnitt definieren wir verschiedene ASCII-Zeichen, Bits im Statusregister und einige Defaultwerte. Außerdem spezifizieren wir den Befehlscode für den als Breakpoint benutzten Befehl TRAP 14.

```
SPACE    EQU    ' '        Leerzeichen      (space)
CR       EQU    $0D        Wagenrücklauf    (carriage return)
LF       EQU    $0A        Zeilenvorschub   (line feed)
BS       EQU    $08        Rückschritt      (backspace)
DEL      EQU    $7F        Zeichen löschen (delete)
TBIT     EQU    7          Trace-Bit im Statusregister
SBIT     EQU    5          Supervisor-Bit im Statusregister
INTSOFF  EQU    $2700      Interrupts aus (Maske für SR)
INTSON   EQU    $2000      Interrupts ein (MASKE für SR)
ISTK     EQU    $2000      Initialwert des Stackzeigers
INTVECS  EQU    0          Lage der Exceptionvektoren
BRKTRP   EQU    $4E4E      Befehlscode TRAP #14
DEFSP    EQU    $2000      Defaultwert für USP
```

Der nächste Abschnitt definiert die Struktur des benutzten Speichers. Wir verwenden keine festen Bereiche als Arbeitsspeicher, sondern stets ein Adreßregister (meistens A6) mit Offset. Zur Behandlung der eingegebenen Zeichen benötigen wir vier Zeiger. Einer zeigt auf einen Zeilenpuffer, und die anderen drei benötigen wir zur Adressierung innerhalb dieses Puffers. Außerdem reservieren wir für die Benutzer-Register und Breakpoint-Adressen Speicherplatz. Den für die Benutzer-Register reservierten Bereich bezeichnen wir mit »**Save-Area**«. Zuletzt geben wir noch verschiedenen Offsets innerhalb des Bereiches für die geretteten Benutzer-Register Namen.

```
BUFWR    EQU    0          Zeiger zum Schreiben
BUFRD    EQU    BUFWR+4    Zeiger zum Lesen
BUFLS    EQU    BUFRD+4    Zeiger auf den Zeilenanfang
BUFBEG   EQU    BUFLS+4    Pufferbereich
BUFEND   EQU    BUFBEG+79  Pufferende
RDUMPD   EQU    BUFEND+1   Raum für 8 Datenregister
RDUMPA   EQU    RDUMPD+32  Raum für 7 Adreßregister
RDUMPSP  EQU    RDUMPA+28  USP
RDUMPSR  EQU    RDUMPSP+4  Benutzer-Statusregister
RDUMPPC  EQU    RDUMPSR+2  Benutzer-Programmzähler
BRKP     EQU    RDUMPPC+4  10 Breakpoints (jeweils 6 Byte)
BFLG     EQU    BRKP+60    Breakpoint-Flag
```

```
RAMEND   EQU      BFLG+2      Ende des Speicherbereiches
*
D_A0     EQU      RDUMPA-RDUMPD      Offset für Register A0
D_A7     EQU      RDUMPSP-RDUMPD     Offset für Register A7
D_SR     EQU      RDUMPSR-RDUMPD     Offset für Statusregister
D_PC     EQU      RDUMPPC-RDUMPD     Offset für Programmzähler
RDSIZE   EQU      (RAMEND-RDUMPD)/4  Größe der Save-Area in
*                                    Vollworten
```

Nachdem wir nunmehr unsere Definitionen abgeschlossen haben, können uns einige Konstantenbereiche anlegen. Der erste Bereich definiert die Exceptionvektoren. Diese werden am Anfang unseres Programmes abgelegt und können auch dort bleiben, falls das Programm ab Adresse Null geladen wird und die Hardware des Computers nicht zwischen Programm- und Datenspeicher unterscheidet. Wird der Monitor zur Ausführung an eine andere Stelle im Speicher geladen, so kopiert er bei seiner Initialisierung diesen Konstantenbereich in den Speicherbereich ab Adresse Null. Die beiden ersten Vollworte definieren die Anfangswerte von Stackzeiger und Programmzähler; diese werden nach einem Reset zum Neustart des Computers benötigt. Dazu muß der Monitor ab Adresse Null geladen sein, oder aber die unteren acht Bytes müssen hardwaremäßig mit den ersten Bytes des Monitors geladen werden. In diesem Fall muß die Startadresse vorher entsprechend bestimmt werden.

```
TS
I_RESET  DC.L     ISTK           Stackzeiger für RESET
         DC.L     SART           Programmzähler für RESET
         DC.L     B_EXCPT-TS     Busfehler
         DC.L     A_EXCPT-TS     Adreßfehler
         DC.L     I_EXCPT-TS     Illegaler Befehl
         DC.L     D_EXCPT-TS     Division durch Null
         DC.L     C_EXCPT-TS     CHK
         DC.L     O_EXCPT-TS     TRAPV
         DC.L     P_EXCPT-TS     Privilegverletzung
I_TRACE  DC.L     T_EXCPT-TS     Trace
         DC.L     X_EXCPT-TS     Unimplementierter Befehl (1010)
         DC.L     Y_EXCPT-TS     Unimplementierter Befehl (1111)
         DS.L     12             noch nicht verwendet
         DC.L     S_EXCPT-TS     unechter Interrupt
*
* Auto-Vektoren für Interrupts
*
         DC.L     INT-TS         Interrupt der Stufe 1
         DC.L     CINT-TS        Interrupt der Stufe 2 (ACIA)
         DC.L     INT-TS         Interrupt der Stufe 3
         DC.L     INT-TS         Interrupt der Stufe 4
         DC.L     INT-TS         Interrupt der Stufe 5
I_INT6   DC.L     INT-TS         Interrupt der Stufe 6
I_INT7   DC.L     INT7-TS        Interrupt der Stufe 7
*
* Trapvektoren
*
         DC.L     TRP-TS         Unerwarteter Trap
         DC.L     TRP-TS         Unerwarteter Trap
         DC.L     TRP-TS         Unerwarteter Trap
```

```
          DC.L    TRP-TS          Unerwarteter Trap
          DC.L    TRP-TS          Unerwarteter Trap
          DC.L    TRP-TS          Unerwarteter Trap
          DC.L    TRP-TS          Unerwarteter Trap
          DC.L    TRP-TS          Unerwarteter Trap
          DC.L    TRP-TS          Unerwarteter Trap
          DC.L    TRP-TS          Unerwarteter Trap
          DC.L    TRP-TS          Unerwarteter Trap
          DC.L    TRP-TS          Unerwarteter Trap
          DC.L    TRP-TS          Unerwarteter Trap
          DC.L    TRP-TS          Unerwarteter Trap
          DC.L    TRP14-TS        Breakpoints
I_BRK     DC.L    TRP15-TS        Ende des Benutzer-Programms
*
INTSIZE   EQU     (I_BRK-I_RESET)/4  Anzahl der festen Vektoren
```

Abschließend spezifizieren wir die Adresse des ACIAs und des verwendeten Speicherbereiches. Wir legen diese Werte im Programmbereich ab, so daß sie später ohne erneute Assemblierung des Programms gepatched werden können (am besten durch den Monitor selbst).

```
ACIA      DC.L    $83FF01         Adresse des ACIAs
RAMBASE   DC.L    $1000           Basisadresse des Speichers
```

8.2 EIN- UND AUSGABE

Nun können wir mit dem eigentlichen Programm beginnen. Die ersten paar Unterprogramme sind für beliebige Verwendung gedacht; sie erwarten daher keinerlei bestimmte Werte in irgendwelchen Registern.

In den später aufgeführten Unterprogrammen, in denen die verschiedenen Monitorbefehle behandelt werden, gehen wir davon aus, daß das letzte eingelesene Zeichen in D0 und in A6 die Basisadresse des Arbeitsspeichers steht.

Das erste Unterprogramm dient zur Behandlung des Interrupts. Da seine Adresse als Auto-Vektor für Interrupts der Stufe 2 verwendet wird, wird es jedesmal aufgerufen, wenn an der Tastatur ein Zeichen eingegeben wird. Beim Aufruf muß sie das Zeichen übernehmen und versuchen, es in einem zyklischen Puffer abzulegen. Ist dies nicht möglich, weil der Puffer voll ist, so wird das Zeichen einfach ignoriert. Handelte es sich dabei um das 'Delete'-Zeichen, so wird das zuletzt eingegebene Zeichen aus dem Puffer entfernt, falls dieser nicht leer ist.

Normalerweise wird das eingegebene Zeichen wieder auf das Terminal ausgegeben. War es das Zeichen 'Delete', so wird die Zeichenfolge 'Rückschritt', 'Leerzeichen', 'Rückschritt' ausgegeben, um das vorhergehende Zeichen vom Bildschirm zu entfernen. War es ein Return, so wird an das Terminal ein Return ausgegeben, gefolgt von einem Zeilenvorschub.

Natürlich ist es lebenswichtig, in der Interruptroutine sämtliche Register zu retten, da sie zu einem beliebigen Zeitpunkt aufgerufen werden kann.

Der zyklische Puffer wird durch drei Zeiger realisiert. BUFWR zeigt auf die augenblickliche Schreibposition, also auf das letzte in den Puffer eingegebene Zeichen. Dieser Zeiger wird normalerweise von dem Unterprogramm INCPTR erhöht, welche den zyklischen Puffer verwaltet. Der Zeiger BUFRD gibt an, wie weit der Puffer bereits durch die Leseroutine geleert wurde. Wenn BUFWR an BUFRD anstößt, ist der Puffer voll und für weitere Zeichen kein Platz mehr vorhanden. Die Leseroutine entfernt nur dann Zeichen aus dem Puffer, wenn eine komplette Zeile eingegeben wurde, damit Zeichen auch wieder gelöscht werden können. Der Zeiger BUFLS zeigt auf den Anfang der akuten Zeile, sowohl zur Unterstützung der Leseroutine als auch um zu gewährleisten, daß kein Zeichen gelöscht wird, wenn die Zeile bereits leer ist. Bei Eingabe von 'Delete' wird der Zeiger BUFWR zyklisch dekrementiert.

Die Ausgabe der Zeichen wird durch das Unterprogramm WRCH durchgeführt, welche das in D0 vorliegende Zeichen auf den ACIA und damit auf das Terminal ausgibt. Obwohl ein ACIA so konfiguriert werden kann, daß er nach vollendeter Ausgabe einen Interrupt verursacht, wird diese Möglichkeit hier nicht verwendet; Ein Interrupt wird nur beim Auftreten eines Zeichens verursacht. Da der Monitor nur ein Programm zur Zeit abarbeiten kann, gibt es nichts Sinnvolles, was getan werden könnte, während wir darauf warten, daß der ACIA eine Übertragung vollendet. Da wir beim Eintreffen eines Zeichens einen Interrupt verursachen, können wir Eingaben zu einem beliebigen Zeitpunkt tätigen.

```
*
* Interrupt-Handler für das Terminal
*
CINT    MOVEM.L  D0/A1/A2/A6,-(SP)
        MOVEA.L  ACIA,A1          Adresse des ACIA
        MOVE.B   2(A1),D0         Zeichen lesen, Interrupt löschen
        ANDI.B   #$7F,D0          Paritätsbit löschen
        MOVEA.L  RAMBASE,A6       Zeiger auf den Anfang des
*                                 Speichers
        MOVEA.L  BUFWR(A6),A2     Zeiger zum Schreiben
        CMP.B    #DEL,D0          'Delete'-Zeichen ?
        BNE.S    CINT2            Nein → normales Zeichen
        CMPA.L   BUFLS(A6),A2     Zeilenanfang ?
        BEQ.S    CINT4            Ja → Zeichen ignorieren
        LEA.L    BUFBEG(A6),A1    Pufferanfang
        CMPA.L   A1,A2            Pufferanfang = Schreibposition ?
        BNE.S    CINT1            Nein → normales Dekrementieren
        LEA.L    BUFEND(A6),A2    Ja → Pufferende verwenden
CINT1   SUBQ.L   #1,D2            Zeiger dekrementieren
        MOVEA.L  ACIA,A1          Adresse des ACIA
        MOVE.B   #BS,D0           Backspace
        BSR.S    WRCH             und ausgeben
        MOVE.B   #SPACE,D0        Dann ein Leerzeichen
        BSR.S    WRCH             und ausgeben
        MOVE.B   #BS,D0           und noch ein Backspace
```

```
        BRA.S    CINT3           Nach Modifizieren des Schreib-
*                                Zeigers ausgeben
CINT2   BSR.S    INCPTR          Zeiger zyklisch erhöhen
        CMPA.L   BUFRD(A6),A2    Schreib-Zeiger = Lese-Zeiger ?
        BEQ.S    CINT4           Ja → Zeichen ignorieren
        MOVE.B   D0,(A2)         Zeichen abspeichern
CINT3   MOVE.L   A2,BUFWR(A6)    Schreib-Zeiger wieder abspeichern
        BSR.S    WRCH1           Zeichen ausgeben
        CMPI.B   #CR,D0          War es ein Return ?
        BNE.S    CINT4           Nein → fertig
        MOVE.L   A2,BUFLS(A6)    Zeiger auf Zeilenanfang
*                                korrigieren
        MOVE.B   #LF,D0          Zeilenvorschub
        BSR.S    WRCH1           Ausgeben
CINT4   MOVEM.L  (SP)+,D0/A1/A2/A6
        RTE                      Rücksprung vom Interrupt
```

In dem folgenden kleinen Unterprogramm gehen wir davon aus, daß A2 einen Zeiger in den Ringpuffer enthält und A6 auf den Anfang des Speicherbereichs zeigt. Es inkrementiert A2 und setzt das Register auf den Pufferanfang zurück, falls das Ende erreicht wurde.

```
INCPTR  MOVE.L   A1,-(SP)
        ADDQ.L   #1,A2           Zeiger inkrementieren
        LEA.L    BUFEND(A6),A1   Zeiger auf das Pufferende
        CMPA.L   A1,A2           Ende erreicht ?
        BNE.S    INC1            Nein → fertig
        LEA.L    BUFBEG(A6),A2   Zeiger auf den Pufferanfang
INC1    MOVE.L   (SP)+,A1
        RTS
```

Mit der Ausgabe beschäfigen sich die folgenden Unterprogramme. Zunächst ist dies das Unterprogramm WRCH, welches das in D0 enthaltene Zeichen an den ACIA ausgibt. Es verwendet ein weiteres Unterprogramm namens WRCH1, das davon ausgeht, daß in A1 die Adresse des ACIAs steht.

```
WRCH    MOVE.L   A1,-(SP)
        MOVEA.L  ACIA,A1         Adresse des ACIAs
        BSR.S    WRCH1           Zeichen ausgeben
        MOVE.L   (SP)+,A1
        RTS
```

Das Unterprogramm WRCH1 wird von den Routinen WRCH, WRITES und CINT aufgerufen, nachdem diese A1 auf die Adresse des Kontrollregisters des ACIAs gesetzt haben. Das Datenregister des ACIAs erscheint im Speicherbereich zwei Byte höher. Dieses Unterprogramm wartet einfach darauf, daß der ACIA Übertragungsbereitschaft anzeigt, und überträgt dann das in D0 enthaltene Zeichen.

```
WRCH1   BTST     #1,(A1)         Ist der ACIA bereit zur Ausgabe?
        BEQ.S    WRCH1           Nein → warten
        MOVE.B   D0,2(A1)        Zeichen übertragen
        RTS
```

Bei den nun folgenden Unterprogrammen handelt es sich um allgemeine Ausgaberoutinen. BLANK gibt einfach ein Leerzeichen, NEWLINE ein von einem Zeilenvorschub gefolgtes Return aus.

```
          BLANK     MOVE.L    D0,-(SP)
          MOVEQ     #SPACE,D0       Leerzeichen
          BSR.S     WRCH            Ausgeben
          BRA.S     NEWL2           Gemeinsamer Rücksprung
*
* Return, gefolgt vom Zeilenvorschub
*
NEWLINE   MOVE.L    D0,-(SP)
          MOVEQ     #CR,D0          Return
          BSR.S     WRCH
          MOVEQ     #LF,D0          Zeilenvorschub
          BSR.S     WRCH
          NEWL2     MOVE.L    (SP)+,D0
          RTS
```

Das nun folgenden Unterprogramm kennen wir (leicht abgeändert) bereits. WRITES gibt bei Aufruf den durch A0 adressierten und mit einer Null begrenzten Text aus. Es ruft nicht WRCH, sondern WRCH1 auf, da es den Zeiger auf den ACIA zu Beginn der Ausgabe ein für allemal setzt.

Abschließend erfolgt ein Sprung nach NEWLINE. Dies ist ein beliebter Programmiertrick. Wenn die letzte Aktion eines Unterprogramms im Aufruf eines anderen Unterprogramms mit anschließendem Rücksprung besteht, ist es einfacher, dieses Unterprogramm direkt anzuspringen. Die Rücksprungadresse für WRITES bleibt auf dem Stack erhalten und wird von dem RTS am Ende von NEWLINE zum Rücksprung verwendet, so daß dieser nicht nach WRITES, sondern zurück in das aufrufende Programm führt.

```
WRITES    MOVEM.L   D0/A1-A1,-(SP)
          MOVEA.L   ACIA,A1         Adresse des ACIAs
WRITES1   MOVE.B    (A0)+,D0        Zeichen aus dem Text holen
          BEQ.S     WRITES2         Null → Ende des Textes
          BSR.S     WRCH1           Zeichen ausgeben
          BRA.S     WRITES1         Nächstes Zeichen
WRITES2   MOVEM.L   (SP)+,D0/A0-A1
          BRA.S     NEWLINE         Neue Zeile und Rücksprung
```

Der letzte Satz Ausgaberoutinen wird zur Ausgabe hexadezimaler Zahlen verwendet. WRHEX4 gibt vier, WRHEX2 zwei und WRHEX1 ein einzelnes Byte aus. WRHEX0 dient zur Ausgabe eines einzelnen Nibbles (4 Bit). Diese Unterprogramme können alle unabhängig voneinander aufgerufen werden; sie resultieren aber alle in der entsprechenden Anzahl von Aufrufen der Routine WRHEX0. Die Werte der Register werden durch das mehrfache SWAPen nicht verändert.

WRHEX4 vertauscht die Registerhälften, um die beiden oberen Bytes durch einen Aufruf von WRHEX2 auszugeben. Anschließend geht es in das Unterprogramm

WRHEX2 über, um dort die beiden unteren Bytes auszugeben.

```
WRHEX4   SWAP     D0              Vertauschen der Registerhälften
         BSR.S    WRHEX2          Die oberen Bytes ausgeben
         SWAP     D0              Erneutes Vertauschen
* Übergang zu WRHEX2
```

WRHEX2 benutzt einen einfachen Trick. Nach Rotation des unteren Wortes wird durch einen Aufruf von WRHEX1 das untere der beiden Bytes ausgegeben. Durch Rückwärtsrotieren wird dann der ursprüngliche Registerinhalt wiederhergestellt und durch Übergang zu WRHEX0 das untere Byte ausgegeben.

```
WRHEX2   ROR.W    #8,D0           Oberes Byte nach unten schieben
         BSR.S    WRHEX1          Ausgabe des oberen Bytes
         ROL.W    #8,D0           Zurückschieben
* ... und Übergang nach WRHEX1 zur Ausgabe des unteren Bytes
```

WRHEX1 ähnelt WRHEX2, jedoch werden hier nur 4 Bit verschoben, um das obere Nibble durch WRHEX0 ausgeben zu können. Dann wird zurückrotiert und das untere Nibble durch einen Übergang nach WRHEX0 ausgegeben.

```
WRHEX1   ROR.B    #4,D0           Oberes Nibble runterschieben
         BSR.S    WRHEX0          und ausgeben
         ROL.B    #4,Do           Zurückschieben
* ... und Übergang nach WRHEX0 zur Ausgabe des unteren Nibbles
```

WRHEX0 schließlich gibt eine einzelne, durch das untere Nibble von D0 spezifizierte hexadezimale Ziffer aus. Der Inhalt von D0 wird dabei nicht verändert.

```
WRHEX0   MOVE.L   D0,-(SP)
         ANDI.B   #$0F,D0         Unteres Nibble maskieren
         ADDI.B   #'0',D0         Zeichen '0' aufaddieren
         CMPI.B   #'9',D0         Ist D0 > '9' ?
         BLS.S    WRHEX01         Nein → ausgeben
         ADDI.B   #'A'-'9'-'1,D0  In einen Buchstaben umwandeln
WRHEX01  BSR.S    WRCH            Hexadezimale Ziffer ausgeben
         MOVE.L   (SP)+,D0
         RTS
```

Der letzte Teil dieses Abschnitts behandelt die Unterprogramme zur Eingabe. Das Gegenstück zur WRCH heißt RDCH und gibt ein Zeichen vom Terminal in D0 zurück. Dabei werden die drei obersten Bytes von D0 gelöscht.

Wie Sie sich sicher erinnern, wird der Ringpuffer von der Interruptroutine verwaltet. Die beiden Zeiger BUFRD und BUFLS zeigen auf das letzte aus dem Puffer gelesene Zeichen bzw. auf den Anfang der Zeile. Wir müssen daher solange warten, bis diese beiden Zeiger nicht mehr auf die gleiche Speicherstelle zeigen, da sonst die Eingabezeile noch nicht durch Return abgeschlossen wurde. Gibt der Benutzer ein **Return** ein, so wird BUFLS auf den Anfang einer neuen Zeile gesetzt, so daß wir nun die eingegebene Zeile verarbeiten können. Wir benutzen hier das bereits bekannte Unterprogramm

INCPTR, um einen Zeiger zyklisch weiterzusetzen.

```
RDCH    MOVEM.L  A2/A6,-(SP)
        MOVEA.L  RAMBASE,A6      Zeiger auf den Datenbereich
        MOVEA.L  BUFRD(A6),A2    Lese-Zeiger holen
RDCH1   CMPA.L   BUFLS(A6),A2    Lese-Zeiger = Anfangszeiger ?
        BEQ.S    RDCH1           Ja → warten
        BSR      INCPTR          Lese-Zeiger erhöhen
        MOVEQ    #0,D0           D0 = 0
        MOVE.B   (A2),D0         Zeichen holen
        MOVE.L   A2,BUFRD(A6)    Lese-Zeiger korrigieren
        MOVEM.L  (SP)+,A6/A2
        RTS
```

Das nächste Unterprogramm liest eine Hexadezimalzahl von der Tastatur ein. Es besitzt zwei Einsprungpunkte. READHEX liest das nächste Zeichen von der Eingabe ein, während READH davon ausgeht, daß das nächste Zeichen bereits gelesen wurde und in Register D0 zur Verfügung steht. Das Ergebnis wird in D1 zurückgegeben, D0 enthält dann das letzte eingelesene Zeichen. Wurde beim Einlesen ein illegales Zeichen, also keine hexadezimale Ziffer, entdeckt, so wird das Z-Bit gelöscht; ansonsten ist es bei Verlassen des Unterprogramms gesetzt. Diese Bedingung kann nachher überprüft und ggf. eine Fehlerbehandlung angesprungen werden.

```
READHEX BSR.S    RDCH            Zeichen holen
READH   CMPI.B   #SPACE,D0       Ist es ein Leerzeichen ?
        BEQ.S    READHEX         Führende Leerzeichen überlesen
        CLR.L    D1              D1 = 0
RDH1    CMPI.B   #'0',D0         Liegt das Zeichen unter '0' ?
        BCS.S    RDH4            Ja → Fehleraussprung
        CMPI.B   #'9',D0         Oberhalb von '9' ?
        BHI.S    RDH2            Ja → Könnte 'A' bis 'F' sein
        SUBI.B   #'0',D0         Zeichen '0' abziehen
        BRA.S    RDH3            Und verarbeiten
RDH2    BSR      LOCASE          In Kleinbuchstaben umwandeln
        CMPI.B   #'a',D0         Unterhalb von 'a' ?
        BCS.S    RDH4            Ja → Fehleraussprung
        CMPI.B   #'f',D0         Oberhalb von 'f' ?
        BHI.S    RDH4            Ja → Fehleraussprung
        SUBI.B   #'a'-10,D0      Richtigen Wert bestimmen
RDH3    ASL.L    #4,D1           D1 = D1 * 16
        ADD.L    D0,D1           D1 = D1 + D0
        BSR.S    RDCH            Nächstes Zeichen lesen
        CMPI.B   #CR,DS0         Return ?
        BEQ.S    RDH4            Ja → Ende der Eingabe
        CMPI.B   #SPACE,D0       Leerzeichen ?
        BNE.S    RDH1            Nein → Bearbeiten
RDH4    RTS
```

Wie Sie sicher bemerkt haben, wird das Unterprogramm beim Label RDH4 verlassen, ohne daß das Z-Bit irgendwo explizit gesetzt oder gelöscht wird. Trat jedoch ein Fehler auf, so wird dieses Label nach einem Vergleich mittels BCS oder BHI angesprungen; das Z-Bit ist dann gelöscht. Beim normalen Aussprung erfolgt der Sprung zu RDH4 mittels BEQ, das Z-Bit ist dann natürlich gesetzt.

8.3 SPRUNGTABELLEN

Im Monitor wollen wir ein Zeichen als Befehl einlesen und dann darauf basierend bestimmte Aktionen durchführen. Dazu benutzen wir das Unterprogramm SEARCH, das in Register D0 ein Zeichen und in A0 die Anfangsadresse einer Sprungtabelle erwartet.

Diese Sprungtabelle enthält für jedes gültige Zeichen die Adresse des anzuspringenden Unterprogramms; außerdem enthält sie eine Default-Adresse, die beim Auftreten eines ungültigen Zeichens angesprungen wird. Jeder Eintrag in der Tabelle besitzt vier Byte, wobei das erste Byte als Markierung für das Tabellenende dient und bei allen außer dem letzten Eintrag den Wert Null hat.

Das zweite Byte enthält das zu der Aktion gehörende Befehlszeichen; dieses Byte wird beim letzten Eintrag ignoriert. Die letzten beiden Bytes dienen zur Angabe des anzuspringenden Unterprogramms, wobei der letzte Eintrag die Default-Adresse für ungültige Zeichen enthält. Um den Monitor relokatibel zu gestalten, enthalten diese beiden Bytes den Offset des Unterprogramms relativ zum Tabellenanfang.

Wir verwenden dieses »Sprungtabellen«-Konzept aus zweierlei Gründen. Zum einen kann ein und dasselbe Programmstück zum Decodieren verschiedener Befehlssätze benutzt werden. Hier können wir SEARCH sowohl zum Decodieren der Monitor- wie auch der Speicherbefehle verwenden. Zum anderen wird so das Einfügen neuer Befehle erleichtert; dazu wird dann nur das entsprechende Unterprogramm sowie ein Eintrag in der Tabelle benötigt.

Das Unterprogramm SEARCH erhält in D0 ein Zeichen, das zunächst in einen Kleinbuchstaben konvertiert wird. Jeder Tabelleneintrag wird daraufhin überprüft, ob er dieses Zeichen enthält; ist dies der Fall, so wird das entsprechende Unterprogramm aufgerufen. Wird das Tabellenende erreicht, ohne daß ein passender Eintrag entdeckt wurde, so wird die dort vorliegende Default-Adresse zum Sprung verwendet.

Durch dieses Unterprogramm werden keinerlei Register verändert, da wir den Stack als Arbeitsspeicher mißbrauchen. Dazu lesen wir zunächst D0 vom vorletzten Stackplatz und speichern dort die Adresse des aufzurufenden Unterprogramms ab. Indem wir nun A0 auf dem normalen Wege wiederherstellen, liegt auf dem Stack als oberster Wert die Adresse des aufzurufenden Unterprogramms, gefolgt von der Rücksprungadresse von SEARCH. Durch den abschließenden RTS erfolgt der Sprung in das Unterprogramm, das dann beim Ausführen seines Rücksprungs wieder direkt in das SEARCH aufrufende Programmstück springt.

```
SEARCH  MOVE.L   D0,-(SP)      D0 retten
        MOVE.L   A0,-(SP)      A0 auch
        BSR.S    LOCASE        D0 in einen Kleinbuchstaben
*                              konvertieren
SRCH1   TST.B    (A0)+         Tabellenende erreicht ?
        BNE.S    SRCH2         Ja
        CMP.B    (A0)+,D0      Zeichen gefunden ?
        BEQ.S    SRCH3         Ja
        ADDQ.L   #2,A0         A0 = A0 + 4 (seit SRCH1)
        BRA.S    SRCH1         Neuer Versuch
SRCH2   ADDQ.L   #1,A0         Unbenutztes Byte für Zeichen
*                              überlesen
SRCH3   MOVEA.W  (A0),A0       Offset vom Tabellenanfang
        ADDA.L   (SP),A0       Plus Tabellenanfang
        MOVE.L   4(SP),D0      D0 wiederherstellen
        MOVE.L   A0,4(SP)      Durch die Adresse ersetzen
        MOVE.L   (SP)+,A0      A0 wiederherstellen
        RTS                    Sprung zu dem Unterprogramm
```

Das folgende kleine Unterprogramm konvertiert das Zeichen in D0 in einen Kleinbuchstaben, falls dies möglich ist. Wenn feststeht, daß eine Umwandlung möglich ist, addieren wir einfach einen Wert auf, der den Abstand zwischen Groß- und Kleinbuchstaben darstellt.

```
LOCASE  CMPI.B   #'A',D0       Enthält D0 einen Großbuchstaben?
        BCS.S    LOC1          Nein → Rücksprung
        CMPI.B   #'Z',D0       Ist er nicht zu groß ?
        BHI.S    LOC1          Doch kein Großbuchstabe
        ADDI.B   #'a'-'A',D0   Umwandlung
LOC1    RTS
```

8.4 INITIALISIERUNG UND ABARBEITEN VON MONITORBEFEHLEN

Das nächste Teilstück unseres Programms behandelt die Initialisierung des Monitors. Das Label START ist der Anfangspunkt für das ganze Programm. Wir haben dessen Adresse im Reset-Vektor abgelegt, so daß dieses Programm bei jedem Reset oder Einschalten des Computers aufgerufen wird (beim Einschalten natürlich nur, wenn Programm und Sprungvektoren noch erhalten sind, etwa in ROM).

Als erstes werden sämtliche Interrupts unterbunden, da noch keine Interrupt-Handler definiert wurden und ein Interrupt somit zu großer Verwirrung führen würde. Danach simulieren wir einen externen Reset, um die Peripheriegeräte in einen definierten Anfangszustand zu versetzen. Falls der Monitor einfach angesprungen und nicht durch einen echten Reset aufgerufen wurde, ist dies besonders wichtig. Anschließend setzen wir noch den von dem Reset nicht betroffenen ACIA in seinen Anfangszustand zurück.

```
START   MOVE.W   #INTSOFF,SR     Interrupts ausschalten,
*                                Supervisormodus einschalten
        RESET
        MOVEA.L  ACIA,A3         Zeiger auf den ACIA
        MOVE.B   #$03,(A3)       ACIA rücksetzen
```

Im nächsten Schritt kopieren wir die Vektortabelle an ihren eigentlichen Platz im untersten Speicher. Ist das Programm ab Adresse Null geladen, so liegt diese Tabelle bereits an ihrer richtigen Stelle. Es schadet aber nichts, wenn wir die Vektortabelle trotzdem kopieren, solange das Programm nicht im ROM vorliegt und die Buskontroll-Logik beim Versuch des schreibenden Zugriffs auf einen ROM einen Busfehler generiert. Bei den meisten Systemen ist dies aber nicht der Fall. Da unser Monitor relokatibel ist, sind die Einträge in der Vektortabelle nicht die absoluten Adresse der anzuspringenden Unterprogramme, sondern nur ihre **Offsets** vom Programmanfang. Beginnt das Programm ab Adresse Null, so stimmen zwar Offset und absolute Adresse überein, aber auch sonst ist dies kein Beinbruch. Wir addieren einfach die Anfangsadresse des Programms zu den einzelnen Offsets und erhalten so die absoluten Adressen.

```
        LEA.L    INTVECS,A1      Zeiger auf die Vektortabelle
        LEA.L    I_REST,A2       Zeiger auf die tatsächliche Lage
        MOVE.L   A2,D2           Zeiger auf den Tabellenanfang
        MOVE.W   #INTSIZE,D0     Anzahl der Einträge
STO     MOVE.L   (A2)+,D1        Tabelleneintrag
        ADD.L    D2,D1           Offset + Basis = Absolut
        MOVE.L   D1,(A1)+        Unten im Speicher ablegen
        DBRA     D0,STO          Bis alle Einträge kopiert wurden
```

Nunmehr setzen wir den Stackzeiger auf seinen ursprünglichen Wert, der durch das erste Vollwort des Reset-Vektors definiert ist. Auch dies ist eigentlich nur dann notwendig, wenn der Monitor einfach angesprungen wurde. Nachdem wir all dies getan haben, können wir auch wieder Interrupts zulassen.

```
        MOVE.L   I_RESET,SP
        MOVE.W   #INTSON,SR
```

Jetzt können wir auch den ACIA sowie die zur Behandlung des Ringpuffers nötigen Zeiger initialisieren. Des weiteren setzen wir A6 als Basiszeiger für den von uns verwendeten Speicherbereich. Dieser Wert muß während der ganzen Ausführungszeit des Monitors in diesem Register erhalten bleiben.

```
        MOVEA.L  RAMBASE,A6      Basiszeiger initialisieren
        LEA.L    BUFBEG(A6),A1   Zeiger auf Pufferanfang
        MOVE.L   A1,BUFWR(A6)    Intialwert: Schreib-Zeiger
        MOVE.L   A1,BUFRD(A6)    Intialwert: Lese-Zeiger
        MOVE.L   A1,BUFLS(A6)    Intialwert: Zeilenanfang
        MOVE.B   #$89,(A3)       ACIA konfigurieren
```

Als nächstes löschen wir den für die Benutzer-Register und Breakpoints vorgesehenen Speicherbereich und initialisieren den Benutzer-Stackzeiger.

```
        LEA.L   RDUMPD(A6),A1   Zeiger auf Save-Area
        MOVE.W  #RDSIZE-1,D0    Größe des zu löschenden Bereichs
CL      CLR.L   (A1)                    Löschen
        DBRA    D0,CL           Bis alles gelöscht
        MOVE.L  #DEFSP,RDUMPSP(A6) USP initialisieren
```

Nun können wir eine Kopfmeldung ausgeben. Dazu lassen wir A0 auf den Text zeigen und rufen WRITES auf.

```
        LEA.L   MESS1,A0        Zeiger auf die Meldung
        BSR     WRITES          Ausgeben
```

Jetzt kommen wir zu der sehr einfachen Hauptausführungsschleife. Hier geben wir zunächst über WRCH einen **Prompt** aus und warten durch Aufruf von RDCH, bis ein Zeichen eingegeben wurde. Dann laden wir A0 mit der Anfangsadresse der Sprungtabelle für die Monitorbefehle und rufen über das Unterprogramm SEARCH das zu dem eingegebenen Befehl gehörende Unterprogramm auf. Nach der Rückkehr aus diesem Unterprogramm springen wir an den Anfang der Schleife zurück; verursachte des Unterprogramm einen Fehler oder startete es ein Benutzer-Programm, so erfolgt der Rücksprung von der Fehler- oder Ausnahmebehandlung direkt wieder nach ST1.

```
ST1     MOVE.B  #'#',D0         Promptzeichen
        BSR     WRCH            ausgeben
ST2     BSR     RDCH            Zeichen einlesen
        LEA.L   COMTAB,A0       Zeiger auf die Sprungtabelle
        BSR     SEARCH          Befehl ausführen
        BRA.S   ST1             und wieder von vorne
```

8.5 DIE UNTERPROGRAMME ZUR BEHANDLUNG DER BEFEHLE

Der Rest des Monitors besteht aus einer Anzahl von Unterprogrammen, die über die Befehlstabelle aufgerufen werden. Bei Aufruf jedes dieser Unterprogramme enthält D0 das entsprechende Zeichen. Außer A6, dem Zeiger auf den Anfang des Arbeitsspeichers, darf jedes Register verändert werden. Alle diese Unterprogramme springen nach Ausführung des Befehls wieder zurück, ausgenommen die, die den Start eines Benutzer-Programms bewirken (T und G). In diesen Fällen erfolgt der Rücksprung in den Monitor durch das Eintreten einer Exception. Das erste Unterprogramm wird angesprungen, falls das eingegebene Zeichen keinen Befehl darstellt. Es wird durch den letzten Eintrag in der Befehlstabelle adressiert. Bei Aufruf gibt es eine Meldung aus und geht dann in ein Standard-Unterprogramm über, in dem die Eingabezeile zu Ende gelesen wird.

```
COMMERR LEA.L   MESS2,A0        Adresse der Meldung
        BSR     WRITES          Ausgeben
```

Das folgende Unterprogramm wird von den meisten anderen aufgerufen, wenn sie ihre Arbeit beendet haben und kurz vor dem Rücksprung in die Hauptschleife

stehen. Es liest nur die Eingabezeile zu Ende und ignoriert alle in ihr noch stehenden Zeichen. D0 muß unbedingt das letzte eingegebene Zeichen enthalten, falls es ein Return war. Dieses Unterprogramm wird auch aufgerufen, falls nur ein Return eingegeben wurde.

```
SKIPNL   CMPI.B   #CR,D0          Letztes Zeichen Return ?
         BEQ.S    SKIPNL1        Ja → zurück
         BSR      RDCH           Zeichen einlesen und ignorieren
         BRA.S    SKIPNL         Bis es ein Return ist
SKIPNL1  RTS
```

Die nächsten beiden Unterprogramme werden beim Entdecken eines Fehlers aufgerufen. Das erste wird verwendet, wenn eine Zahl erwartet, aber nicht gefunden wurde. Da es SKIPNL aufruft, muß in D0 das letzte gelesene Zeichen stehen.

```
NUMERR   LEA.L    MESS3,A0        Fehlermeldung
         BSR      WRITES          Ausgeben
         BRA.S    SKIPNL          Rest der Zeile ignorieren
```

Das nächste Unterprogramm ist mit NUMERR fast identisch. Es wird aufgerufen, wenn ein ungültiger Speicherbefehl eingegeben wurde.

```
MEMERR   LEA.L    MESS4,A0        Fehlermeldung
         BSR      WRITES          Ausgeben
         BRA.S    SKIPNL          Rest der Zeile ignorieren
```

8.6 AUSGABE UND MODIFIKATION VON REGISTERN

Die folgenden Unterprogramme dienen zur Ausgabe und Änderung der Benutzer-Register. Das erste, REG, wird durch Eingabe des R-Befehls aufgerufen. Es gibt einfach die Inhalte aller Benutzer-Register aus. Es hat zwei Einsprungpunkte: REG wird bei Eingabe von 'R' aufgerufen, und REGX dient zur Ausgabe der Registerinhalte beim Tracen oder beim Auftreten eines Breakpoints. Der einzige Unterschied besteht darin, daß beim Einsprung über REG der Rest der Zeile überlesen wird.

In den ersten Zeilen werden mittels WRHEX4 und WRHEX2 die Inhalte von Benutzer-Stackzeiger und -Statusregister ausgegeben. Hierbei gehen wir davon aus, daß in A6 die Startadresse des Arbeitsspeichers steht.

```
REG      BSR      SKIPNL          Rest der Zeile überlesen
REGX     MOVEQ    #'P',D0         Kennbuchstabe des Registers
         BSR      WRCH            Ausgeben
         MOVEQ    #'C',D0         Zweiter Buchstabe
         BSR      WRCH            Und raus damit
         BSR      BLANK           Dann noch ein Leerzeichen
         MOVE.L   RDUMPPC(A6),D0  Inhalt des Stackzeigers
         BSR      WRHEX4          Ebenfalls ausgeben
         BSR      BLANK           Und noch ein Leerzeichen
```

```
        MOVEQ     #'S',D0          Schon wieder ein Registername
        BSR       WRCH
        MOVEQ     #'R',D0          Noch ein Buchstabe
        BSR       WRCH
        BSR       BLANK            Und schon wieder ein Leerzeichen
        MOVE.W    RDUMPSR(A6),D0   Inhalt des Statusregisters
        BSR       WRHEX2           Nur das untere Wort ausgeben
        BSR       NEWLINE          Neue Zeile
```

In den nächsten Zeilen werden die Inhalte der Daten- und Adreßregister ausgegeben. Da es bei der Ausgabe dieser beiden Registertypen keine Unterschiede gibt (da in beiden Fällen die Werte der Save-Area entnommen werden), verwenden wir das Unterprogramm REG1, das in D1 die Kennung des Registertyps (D oder A) und in A3 einen Zeiger auf die Save-Area der entsprechenden Register erhält. Nach dem ersten Aufruf von REG1 zur Ausgabe der Datenregister zeigt A3 auf die Save-Area der Adreßregister, so daß zu deren Ausgabe nur noch die Änderung der Kennung und ein Übergang nach REG1 vonnöten ist.

```
        LEA.L     RDUMPD,(A6),A3 Zeiger auf die Datenregister
        MOVEQ     #'D',D1        Buchstabe für Registernamen
        BSR.S     REG1           Registersatz ausgeben
        MOVEQ     #'A',D1        Buchstabe für Registernamen
* ... und Übergang zu REG1
```

REG1 gibt die acht Benutzer-Register aus, die in dem durch A3 adressierten Speicherbereich vorliege. Die Register werden dabei durch den in D0 liegenden Buchstaben bezeichnet. WRHEX0 wird zur Ausgabe der Registernummer, WRHEX4 zur Ausgabe des Wertes aufgerufen. Das Register A3 wird bei der Ausgabe der Werte erhöht, so daß es nach Abschluß der Ausgabe hinter den ausgegebenen Bereich zeigt, nach Ausgabe der Datenregister also auf die Adreßregister. Bei der Ausgabe werden jeweils vier Register in einer Zeile zusammengefaßt.

```
REG1    MOVEQ     #0,D2          Registernummer
REG2    MOVE.B    D1,D0          Registerkennung ('D' oder 'A')
        BSR       WRCH
        MOVE.B    D2,D0          Registernummer
        BSR       WRHEX0         Als Ziffer ausgeben
        BSR       BLANK          Leerzeichen
        ADDQ.B    #1,D2          Registernummer erhöhen
        MOVE.L    (A3)+,D0       Wert des Registers
        BSR       WRHEX4         Und ausgeben
        CMP.B     #8,D2          Fertig ?
        BEQ.S     REG3           Ja
        BSR       BLANK          Noch ein Leerzeichen
        CMP.B     #4,D2          Register 4 als nächstes ?
        BNE.S     REG2           Nein → Noch ein Register
        BSR       NEWLINE        Sonst neue Zeile beginnen
        BRA.S     REG2           Und Register 4 - 7 ausgeben
REG3    BRA       NEWLINE        Abschließend neue Zeile und
*                                Rücksprung
```

Zur Ausgabe und Modifikation von Daten- und Adreßregistern gibt es die beiden Befehle 'A' und 'D', die durch eine Registernummer spezifiziert werden müssen. Folgt anschließend noch eine Zahl, so wird diese eingelesen und dem Register zugewiesen. Ansonsten wird einfach der Inhalt des entsprechenden Registers ausgegeben.

Die Unterprogramme verwenden gemeinsamen Code, nachdem sie A3 auf den Anfang der Save-Area für die Daten- bzw. Adreßregister gesetzt haben.

```
SETD    LEA.L   RDUMPD(A6),A3   Adresse der Datenregister
        BRA.S   SETR            Sprung zum gemeinsamen Code
*
* Adreßregister
*
SETA    LEA.L   RDUMPA(A6),A3   Adresse der Adreßregister
```

Im gemeinsamen Programmabschnitt wird zunächst die auf den Buchstaben 'D' bzw. 'A' folgende Registernummer eingelesen. Dazu wird das Unterprogramm READHEX aufgerufen und die eingelesene Zahl auf ihre Gültigkeit hin überprüft. Handelt es sich nicht um eine gültige Registernummer, so erfolgt ein Sprung nach SETRE. War der Wert dagegen korrekt, so wird er mit vier multipliziert und ergibt dann einen Offset für den Registerbereich.

```
SETR    BSR     READHEX
        BNE.S   SETRE           Registernummer erwartet
        TST.L   D1              Untere Grenze prüfen
        BLT.S   SETRE           Fehler
        CMP.L   #7,D1           Obere Grenze prüfen
        BGT.S   SETRE           Auch ein Fehler
        MOVE.W  D1,D3           In D3 abspeichern
        ASL.W   #2,D3           D3 = D3 * 4
```

Der nächste Programmabschnitt wird auch vom 'P'-Befehl verwendet, der den Benutzer-Programmzähler ausgibt bzw. verändert. Zunächst wird dabei geprüft, ob das letzte eingelesene Zeichen (liegt nach Aufruf von READHEX in D0 vor) ein Return war. Folgt der Spezifikation des Registers kein neuer Wert, so wird der alte durch Einsprung bei SETR2 ausgegeben.

War die Eingabezeile mit der Registerangabe noch nicht beendet, so wird der folgende Wert durch Einsprung beim Label READH im Unterprogramm READHEX eingelesen. Wird dabei ein Fehler entdeckt, so erfolgt anschließend ein Sprung zum Unterprogramm NUMERR. Ansonsten wird der Wert an der durch den Offset in Register D3 und der Basisadresse in A3 adressierten Speicherstelle abgelegt. Durch Sprung nach SKIPNL springt das Unterprogramm dann wieder in die Hauptschleife des Monitors zurück.

```
SETR1   CMP.B   #CR,D0          Letztes Zeichen Return ?
        BEQ.S   SETR2           Ja → Registerinhalt ausgeben
        BSR     READH           Wert einlesen
        BNE     NUMERR          Hexzahl erwartet
        MOVE.L  D1,0(A3,D3.W)   Wert eintragen
        BRA     SKIPNL          Rest überlesen und Rücksprung
```

Wurde kein neuer Wert eingegeben, so wird das Label SETR2 angesprungen. In diesem Fall wird der momentane Wert des Registers aus der Save-Area geholt und ausgegeben.

```
SETR2    MOVE.L   0(A3,D3.W)      Registerinhalt holen
         BSR      WRHEX4          Als Vollwort ausgeben
         BRA      NEWLINE         Neue Zeile und Rücksprung
```

Bei Eingabe des Befehls 'P' wird das Unterprogramm SETP aufgerufen, in dem der Wert des Benutzer-Programmzählers ausgegeben bzw. modifiziert wird. Dieser Wert kann auch von den Befehlen G und T verändert werden. A3 wird einfach mit dem Zeiger auf die Save-Area für den Programmzähler geladen und der durch die Registernummer gegebene Offset auf Null gesetzt. Dadurch wird bei Einsprung in den gemeinsamen Code bei SETR1 die richtige Stelle adressiert. Vor diesem Sprung wird das nächste Zeichen von der Eingabe nach D0 gelesen, da dies bei SETR1 ausgewertet wird.

```
SETP     LEA.L    RDUMPC(A6),A3   Save-Area für Programmzähler
         CLR.W    D3              Offset = 0
         BSR      RDCH            Nächstes Zeichen lesen
         BRA.S    SETR1           In den gemeinsamen Code springen
```

Zur Ausgabe und Modifikation des Statusregisters dient der Befehl 'S'. Das entsprechende Unterprogramm kann den bisher schon ausgiebig genutzten Code bei SETR1 nicht verwenden, da das Statusregister nur Wortgröße hat. Der evtl. ins Statusregister übernommene Wert wird vorher in keinster Weise überprüft. Wenn der Benutzer das Trace-Bit setzt, läuft sein Programm anschließend im Tracemodus ab. Auch das Supervisor-Bit, das beim Start eines Benutzer-Programms nicht gesetzt sein darf, kann durch diesen Befehl eingeschaltet werden.

```
SETS     BSR      RDCH            Nächstes Zeichen lesen
         CMP.B    #CR,D0          War es Return ?
         BEQ.S    SETS1           Nein → momentanen Wert ausgeben
         BSR      READH           Hexzahl nach D0 einlesen
         BNE      NUMERR          Fehler beim Einlesen der Zahl
         MOVE.W   D1,RDUMPSR(A6)  Statusregister modifizieren
         BRA      SKIPNL          Rücksprung
SETS1    MOVE.W   RDUMPSR(A6),D0  Momentanes Statusregister
         BSR      WRHEX2          Als Wort ausgeben
         BRA      NEWLINE         Neue Zeile und Rücksprung
```

Als letztes Unterprogramm dieses Abschnitts folgt nun SETRE, das zur Ausgabe einer Fehlermeldung verwendet wird.

```
SETREW   LEA.L    MESS6,A0        Zeiger auf Meldung
         BSR      WRITES
         BRA      SKIPNL          Rest überlesen und Rücksprung
```

8.7 DER START EINES BENUTZER-PROGRAMMS

In diesem Abschnitt implementieren wir die Befehle T, G und C, die zum Start eines Benutzer-Programms verwendet werden. Der Einsprung TGO ist dabei für den Befehl T, und der Einsprung GO für den Befehl G gedacht; beim Einsprungpunkt TGO wird nur das Trace-Bit im geretteten Benutzer-Statusregister gesetzt. Der Einsprungpunkt CONT dient für den Befehl C, welcher die Weiterverarbeitung eines aufgrund eines Breakpoints unterbrochenen Programms veranlaßt.

Als erstes wird geprüft, ob hinter dem Befehl eine Startadresse für das Benutzer-Programm angegeben wurde. Ist dies nicht der Fall, so erfolgt ein Sprung zum Label GO1. Ansonsten wird mittels READH der Einsprungpunkt für das Benutzer-Programm eingelesen. Tritt dabei kein Fehler auf, so wird der gerettete Benutzer-Programmzähler auf den neuen Wert gesetzt.

```
* Trace-Modus einschalten

TGO     BSET    #TBIT,RDUMPSR(A6)

* Normaler Modus

GO      BSR     RDCH            Nächstes Zeichen lesen
        CMP.B   CR,D0           War es ein Return ?
        BEQ.S   GO1             Ja → Programmstart
        BSR     READH           Startadresse einlesen
        BNE     NUMERR          Fehler beim Einlesen
        MOVE.L  D1,RDUMPPC(A6)  Programmzähler neu setzen
```

Bei Label GO1 wird dann das Benutzer-Programm gestartet. Dieses Codestück kann auch aufgerufen werden, wenn das Programm durch Tracen unterbrochen wurde und der nächste Befehl zur Ausführung ansteht.

Zuerst muß überprüft werden, ob im Benutzer-Statusregister das Supervisor-Bit gesetzt ist. Ist dies der Fall, so wird das Programm nicht gestartet, sondern eine Fehlermeldung ausgegeben.

```
GO1     BTST    #SBIT,RDUMPSP(A6)  Supervisor-Bit gesetzt ?
        BNE.S   GOERR                    Ja → Fehler
        CLR.B   BFLG(A6)           Breakpoints löschen
```

Als nächstes werden die Breakpoints ins Benutzer-Programm eingebaut. Breakpoints werden nur dann eingebaut, wenn das Programm zum Ablauf gebracht werden soll, so daß der Benutzer bei der Untersuchung seines Codes diesen unverändert vorfindet. Die Adressen der zu installierenden Breakpoints liegen in der Tabelle BRKP vor. Diese verwendet sechs Byte je Eintrag, wobei die ersten vier Bytes die Adresse des Breakpoints beinhalten. Diese hat den Wert Null, falls der entsprechende Breakpoint nicht gesetzt ist. Die letzten beiden Bytes enthalten den Code, der eigentlich an dieser Adresse vorlag und durch den Befehl TRAP 14 ersetzt wurde. Wir laden die Adresse des

zu setzenden Breakpoints nach D1, um sie auf Null zu überprüfen. (MOVEA ändert schließlich keine Condition-Codes.) Da wir später mit diesem Wert adressieren wollen, benötigen wir ein leeres Adreßregister, das wir durch Löschen von A2 erhalten. Die Adressierung 0(A2,D1.L) erlaubt uns also, D1 wie ein Adreßregister zu benutzen.

Eine zusätzliche Schwierigkeit ergibt sich, wenn wir den Befehl an der Stelle, wo der Breakpoint liegt, ausführen wollen. Normalerweise werden beim Erreichen eines Breakpoints die nächsten Befehle im Tracemodus ausgeführt, wobei dann natürlich ein Breakpoint nicht als solcher behandelt werden soll. Stattdessen ist dann die normale Ausführung des dort liegenden Befehls erwünscht. Daher prüfen wir, ob die Adresse eines Breakpoints gleich dem momentanen Benutzer-Programmzähler ist. In diesem Fall fügen wir dann keinen Breakpoint in den Code ein. Wir kopieren aber trotzdem den originalen Code in die entsprechenden Stellen, damit später beim Entfernen der Breakpoints und der damit verbundenen Wiedererstellung des Codes keine Sonderfälle beachtet werden müssen.

Um nach einem Breakpoint die Abarbeitung des Programms wiederaufzunehmen, wird der Befehl 'C' verwendet. Dabei nutzen wir die Tatsache aus, daß an der momentanen Position des Programmzählers kein Breakpoint installiert wird. Bei Aufruf des C-Befehls setzen wir daher zunächst das Trace-Bit und springen dann zum Label CGO. Dadurch führen wir den durch den Breakpoint ersetzten Befehl aus und geben die Kontrolle aufgrund der Trace-Exception sofort wieder an Monitor zurück. Bei der Bearbeitung des C-Befehls haben wir das Flag BFLG gesetzt. Der Exception-Handler prüft dieses Flag, und wenn es gesetzt ist, startet er das Programm durch Aufruf der G-Routine neu. Dadurch ist sichergestellt, daß der Breakpoint wieder in den Code eingebaut wird, um ggf. ein weiteres Mal das Programm zu unterbrechen. Das Flag BFLG wird dabei gelöscht, so daß diese besondere Behandlung der Exception bei der Angabe von T oder G nicht durchgeführt wird.

```
CGO     MOVEA.L  RDUMPPC(A6),A4      Benutzer-Programmzähler
        LEA.L    BRKP(A6),A1         Breakpoint-Tabelle
        MOVEQ    #9,D0               Zähler setzen
        SUBA.L   A2,A2               A2 = 0GO2
        MOVE.L   (A1)+,D1            Adresse des Breakpoints
        BEQ.S    GO3                 Null → Breakpoint nicht
*                                    gesetzt
        MOVE.W   0(A2,D1.L),(A1)     Ursprünglichen Befehl retten
        CMPA.L   D1,A4               Breakpoint-Adresse = PC ?
        BEQ.S    GO3                 Ja → kein Breakpoint
        MOVE.W   #BRKTRP,0(A2,D1.L)  Breakpoint installieren
GO3     ADDQ.L   #2,A1               A1 = A1 + 6 (seit GO2)
        DBRA     D0,GO2              Nächster Breakpoint
```

Nun können wir das Benutzer-Programm starten. Dazu laden wir uns zunächst den geretteten Benutzer-Stackzeiger über das Register A0 in den USP. Dieser Umweg ist nötig, da nur ein Adreßregister ins USP geladen werden kann.

Als nächstes werden Benutzer-Programmzähler und -Statusregister geholt und für den folgenden RTE-Befehl auf dem Stack abgelegt. Dann laden wir mittels MOVEM die Benutzer-Register mit den in der Save-Area abgelegten Werten und führen dann den RTE-Befehl zum Start des Benutzer-Programms aus. Da das Supervisor-Bit des auf dem Stack liegenden Statusregisters gelöscht ist, läuft das Programm dann im Benutzermodus ab. Der Rücksprung zum Monitor erfolgt nur durch das Eintreten einer Exception, daher haben wir TRAP 15 zum Rücksprung in den Monitor bei Ende des Benutzer-Programms reserviert.

```
MOVE.L   RDUMPSP(A6),A0       Benutzer-Stackzeiger holen
MOVE.L   A0,USP               und eintragen
MOVE.L   RDUMPPC(A6),-(SP)    PC auf dem Stack abspeichern
MOVE.L   RDUMPSR(A6),-(SP)    SR auf dem Stack abspeichern
MOVEM.L  RDUMPD(A6),D0-D7/A0-A6 Benutzer-Register setzen
RTE                           Start des Benutzer-Programms
```

Der Befehl 'C' wird zur Wiederaufnahme des Programms nach einem Breakpoint verwendet. Wie üblich wird das Supervisor-Bit im geretteten Statusregister geprüft, das Trace-Bit eingeschaltet und zum Start des Benutzer-Programms zum Label CGO gesprungen. Da der Benutzer-Programmzähler zu diesem Zeitpunkt auf einen Breakpoint zeigt, wird dieser vor dem Start nicht in den Code eingefügt. Das Flag BFLG setzen wir vorher mittels Scc mit cc = TRUE. Dieses Flag wird im Exception-Handler verwendet, um zwischen normalem Tracen und dem nach einem Breakpoint eingeschalteten Tracemodus zu unterscheiden. Im zweiten Fall ersetzen wir dann einfach den Befehl im Code durch den Breakpoint, löschen ggf. das Trace-Bit und fahren mit der Programmabarbeitung fort. Dadurch erscheinen die Breakpoints dem Benutzer die ganze Zeit über gesetzt.

```
CONT     BSR      SKIPNL              Rest der Zeile überlesen
         BTST     #SBIT,RDUMPSR(A6)   Supervisor-Bit gesetzt ?
         BNE.S    GOERR               Ja → Fehler
         BSET     #TBIT,RDUMPSR(A6)   Trace-Bit setzen
         ST       BFLG(A6)            Flag auf $FF setzen
         BRA.S    CGO                 Benutzer-Programm starten
```

GOERR wird angesprungen, falls das Supervisor-Bit im geretteten Statusregister gesetzt ist. Eine entsprechende Fehlermeldung wird ausgegeben und in den Monitor zurückgesprungen.

```
GOERR    LEA.L    MESS7,A0            Fehlermeldung
         BSR      WRITES              Ausgeben
         BRA      SKIPNL              Zeile überlesen
```

8.8 AUSGABE UND VERÄNDERUNG VON SPEICHERSTELLEN

Der nächste Satz von Unterprogrammen dient zur Ausgabe und Modifikation des Speichers. Dieser Modus des Monitors wird durch Eingabe des Befehls 'M' aktiviert.

Die angegebene Speicherstelle wird dabei geöffnet und ihr augenblicklicher Wert ausgegeben. Anschließend können die Speicherbefehle des Monitors benutzt werden, um den Wert der geöffneten Stelle auszugeben, zu ändern, eine andere Stelle zu öffnen oder wieder in den normalen Modus zurückzukehren.

Speicherstellen können als Bytes, Worte oder Vollworte geöffnet werden, wobei anfangs die Bytegröße voreingestellt ist. Register D2 enthält die Größe der geöffneten Stelle: 1 für Byte, 2 für Wort und 4 für Vollwort. Die aktuelle Adresse speichern wir in A3; im folgenden ersten Teil geben wir sie aus.

```
MEM     BSR     READHEX     Adresse einlesen
        BNE     NUMERR      Fehler beim Einlesen
        MOVEA.L D1,A3       Adresse nach A3
        MOVEQ   #1,D2       Voreinstellung: Byte
MEM1    MOVE.L  A3,D0       Adresse nach D0
        BSR     WRHEX4      Als Vollwort ausgeben
        BSR     BLANK       Und ein Leerzeichen
```

Als nächstes geben wir den an dieser Stelle vorliegenden Wert aus. Die Größe des Wertes entnehmen wir dabei dem Register D2. Entsprechend seinem Inhalt rufen wir WRHEX1, WRHEX2 oder WRHEX4 auf.

```
        CMP.B   #2,D2       Welche Größe ?
        BLT.S   MEMB        < 2 → Byte
        BEQ.S   MEMW        = 2 → Wort
        MOVE.L  (A3),D0     Vollwort bei A3
        BSR     WRHEX4      Ausgeben
        BRA.S   MEMQ
MEMW    MOVE.W  (A3),D0     Wort bei A3
        BSR     WRHEX2      Ausgeben
        BRA.S   MEMQ
MEMB    MOVE.B  (A3),D0     Byte bei A3
        BSR     WRHEX1      Ausgeben
MEMQ    MOVE.B  #'?',D0     Fragezeichen
        BSR     WRCH        Und ausgeben
```

Als nächstes prüfen wir, ob nun ein neuer Wert oder ein Speicherbefehl eingegeben wird. Da das Unterprogramm READHEX das Einlesen beim Auftreten eines Zeichens, das keine hexadezimale Ziffer darstellt, einstellt, können wir es auch zum Einlesen des Befehls-Buchstabens verwenden, der dann beim Rücksprung in D0 steht. Wir geben dann ausnahmsweise nicht direkt eine Fehlermeldung aus, sondern prüfen, ob es sich um einen legalen Speicherbefehl handelt.

```
        BSR     READHEX     Wert einlesen
        BNE.S   MEM2        Keine Zahl → Befehl ?
        BSR     SKIPNL      Rest der Zeile überlesen
```

Wird der Branch nicht ausgeführt, so wurde eine Hexzahl eingelesen; wir müssen also den Wert der geöffneten Speicherstelle ändern. Danach wollen wir den Inhalt dieser Speicherstelle nicht erneut ausgeben, da dies bei Registern in I/O-Baustei-

nen wie ACIAs zu Komplikationen führen kann. Wir werden nachher noch ein Unterprogramm namens NMEM kennenlernen, durch das die nächste Speicherstelle geöffnet wird; dieses müssen wir nun aufrufen. Anstatt nun für alle drei Fälle (Byte, Wort und Vollwort) einen Unterprogrammaufruf und anschließenden Sprung nach MEM1 zu bewerkstelligen, legen wir die Adresse von MEM1 mittels PEA auf dem Stack ab. Danach können wir NMEM anspringen und werden uns nach der Ausführung des Rücksprungs bei MEM1 wiederfinden.

```
        PEA.L   MEM1        Adresse von MEM1 auf den Stack
        CMP.B   #2,D2       Wieder mal die Größe testen
        BLT.S   MEMBW       Byte
        BEQ.S   MEMWW       Wort
        MOVE.L  D1,(A3)     Vollwort abspeichern
        BRA.S   NMEM        Nächster Wert, Sprung nach MEM1
MEMWW   MOVE.W  D1,(A3)     Wort abspeichern
        BRA.S   NMEM        Nächster Wert, Sprung nach MEM1
MEMBW   MOVE.B  D1,(A3)     Byte abspeichern
        BRA.S   NMEM        Nächster Wert, Sprung nach MEM1
```

Wurde keine gültige Hexzahl eingelesen, so gehen wir zunächst von der Eingabe eines Befehls aus. Der entsprechende Buchstabe steht dann in D0. Zur Auswahl der richtigen Aktion verwenden wir das bereits bekannte Unterprogramm SEARCH, dem wir diesmal eine andere Sprungtabelle namens MEMTAB in A0 übergeben. Diese ähnelt der in der Hauptschleife des Monitors auftretenden Tabelle COMTAB, enthält aber die Adressen und Buchstaben für die Speicherbefehle. Nach dem Rücksprung aus einem dieser Unterprogramme befinden wir uns nach wie vor im Speichermodus. Bei Eingabe eines Punkts (.) verlassen wir den Speichermodus und springen wieder wieder in den normalen Monitor zurück, nachdem wir vorher den Rest der aktuellen Eingabezeile überlesen haben.

```
MEM2    CMPI.B  '.',D0     Beendung des Speichermodus ?
        BEQ.S   MEM3        Ja
        LEA.L   MEMTAB,A0   Speicherbefehle
        BSR     SEARCH      Unterprogramm suchen und aufrufen
        BSR     SKIPNL      Zeile überlesen
        BRA.S   MEM1        Zurück zum Anfang
MEM3    BRA     SKIPNL      Zeile überlesen, zurück zum Monitor
```

Die nun folgenden Unterprogramme werden fast alle durch die Tabelle MEMTAB adressiert und somit von SEARCH aufgerufen. Das erste öffnet die nächste Speicherstelle; es wird bei Eingabe eines **Returns** sowie nach der Eingabe und Abspeicherung eines neuen Wertes im Speicher aufgerufen. Das Format der zu öffnenden Speicherstelle steht nach wie vor in D2 verschlüsselt, und die Adresse der augenblicklich geöffneten Speicherstelle ist in A3 abgelegt.

```
NMEM    ADDA.L  D2,A3      Nächste Speicherstelle
        RTS
```

Das folgende Programmstück ist NMEM sehr ähnlich, es öffnet die vorherige Speicherstelle.

```
PMEM    SUBA.L   D2,A3       Vorherige Speicherstelle
        RTS
```

Die Größe der geöffneten Speicherstelle wird durch die folgenden Unterprogramme geändert. Beim Rückschritt zur Bytegröße bedeutet dies nur die Veränderung von D2.

```
SETB    MOVEQ    #1,D2       Bytegröße
        RTD
```

Bei der Änderung auf Wort- oder Vollwortgröße muß zur Vermeidung eines Adreßfehlers die aktuelle Adresse in A3 geradzahlig sein. Dies wird durch Aufruf des Unterprogramms CHKEVEN gewährleitstet; dieses springt nur dann zum rufenden Programm zurück, wenn A3 eine gerade Zahl enthält; anderenfalls springt es direkt zu dem SETW bzw. SETL aufrufenden Programm zurück.

```
* Wortgröße einstellen

SETW    BSR.S    CHKEVEN     Ist die Adresse gerade ?
        MOVEQ    #2,D2       Ja → Wortgröße einstellen
        RTS
* Vollwortgröße einstellen

SETL    BSR.S    CHKEVEN     Ist die Adresse gerade ?
        MOVEQ    #4,D2       Ja → Vollwortgröße einstellen
        RTS
```

Nun benötigen wir nur noch das Unterprogramm zum Testen einer Adresse. Scheitert der Test, so wird eine Fehlermeldung ausgegeben, die eigentliche Rücksprungadresse ignoriert und die nächste Adresse vom Stack genommen. Dies ist dann die Rücksprungadresse des Unterprogramms, das CHKEVEN aufrief, in unserem Fall also immer MEM.

```
CHKEVEN MOVE.L   A3,D0       D0 = aktuelle Adresse
        BTST     #0,D0       Unteres Bit testen
        BEQ.S    CHK1        0 → Adresse gerade
        LEA.L    MESS5,A0    Fehlermeldung
        BSR      WRITES
        LEA.L    4(SP),SP    Rücksprungadresse übergehen
CHK1    RTS                  Rücksprung oder Sprung nach MEM
```

8.9 BREAKPOINTS

In den nächsten Unterprogrammen behandeln wir das Setzen, Löschen und Auflisten der Breakpoints. Im Unterprogramm GOC haben wir bereits erklärt, daß eine

Breakpoint-Tabelle vorliegt, in der für jeden Breakpoint in sechs Bytes die Adresse und der ursprünglich dort liegende Code gespeichert sind.

In diesem Abschnitt befassen wir uns ausschließlich mit der Manipulation der Adressen der Breakpoints. Der Befehl B alleine gibt alle gesetzten Breakpoints aus; nicht gesetzte Breakpoints sind daran zu erkennen, daß sie im Adreßfeld den Wert Null haben.

```
BRK     LEA     BRK(A6),A1      Zeiger auf die Breakpoint-Tabelle
        BSR     RDCH            Nächstes Zeichen einlesen
        CMP.B   #CR,D0          Return ?
        BNE.S   BRK1            Nein → komplexere Aktion nötig

* Gesetzte Breakpoints ausgeben

        MOVEQ   #0,D1           Zähler initialisieren
BRK0    TST.L   (A1)            Breakpoint gesetzt ?
        BEQ.S   BRK01           Nein → Eintrag übergehen
        MOVE.B  D1,D0           D0 = Nummer des Breakpoints
        BSR     WRHEX0          Ausgeben
        BSR     BLANK           Leerzeichen
        MOVE.L  (A1),D0         D0 =  Adresse des Breakpoints
        BSR     WRHEX4          Ausgeben
        BSR     NEWLINE         Neue Zeile
BRK01   ADDQ.L  #6,A1           Zeiger erhöhen
        ADDQ.B  #1,D1           Nummer erhöhen
        CMP.B   #9,D1           Ende erreicht ?
        BLE.S   BRK0            Nein → weiter
        RTS
```

Wurde der Befehl nicht mit einem Return abgeschlossen, so muß dem 'B' die Nummer eines Breakpoints folgen. Da wir RDCH aufriefen, enthält D0 das eingelesene Zeichen; daher können wir die Nummer durch Einsprung bei READH ermitteln.

```
BRK1    BSR     READH           Hexzahl lesen
        BNE.S   BRKE            Fehler beim Einlesen
        TST.L   D1              Untere Grenze prüfen
        BLT.S   BRKE            < 0 → zu klein
        CMP.L   #9,D1           > 9 ?
        BGT.S   BRKE            Ja → zu groß
```

Nachdem nun eine korrekte Nummer vorliegt, müssen wir den zugehörigen Offset bestimmen. Da jeder Eintrag sechs Byte benötigt, benutzen wir einfach MULS. Dies können wir tun, da glücklicherweise nur zehn Einträge in der Tabelle erlaubt sind, so daß uns die Beschränkungen der Operanden von MULS nicht berühren. Danach prüfen wir, ob nach der Nummer des Breakpoints noch weitere Werte eingegeben wurden. Wenn nicht, so löschen wir einfach den Breakpoint; ansonsten setzen wir einen neuen.

```
MULS    #6,D1           D1 = D1 * 6 (Offset)
        ADDA.L  D1,A1           Zeiger auf den Eintrag
        CMP.B   #CR,D0          Kommt noch was ?
        BNE.S   BRK2            Ja
```

Nun löschen wir einfach den durch A1 adressierten Breakpoint.

```
        CLR.L   (A1)            Breakpoint löschen
        RTS
```

Andernfalls müssen wir den angegebenen Wert einlesen und den entsprechenden Eintrag in der Tabelle ändern.

```
BRK2    BSR     READH           Wert einlesen
        BNE     NUMERR          Fehler beim Einlesen
        MOVE.L  D1,(A1)         Adresse ablegen
        BRA     SKIPNL
```

Bleibt noch die Fehlermeldung im Falle einer ungültigen Breakpoint-Nummer.

```
BRKE    LEA.L   MESS8,A0        Fehlermeldung
        BSR     WRITES
        BRA     SKIPNL
```

8.10 BEHANDLUNG VON EXCEPTIONS

Dieser Programmabschnitt beschäftigt sich mit der Behandlung aller möglichen auftretenden Exceptions, seien es Fehler, Traps oder Interrupts. Als Standard-Aktion wird nach dem Retten der Register in die Save-Area eine Fehlermeldung ausgegeben.

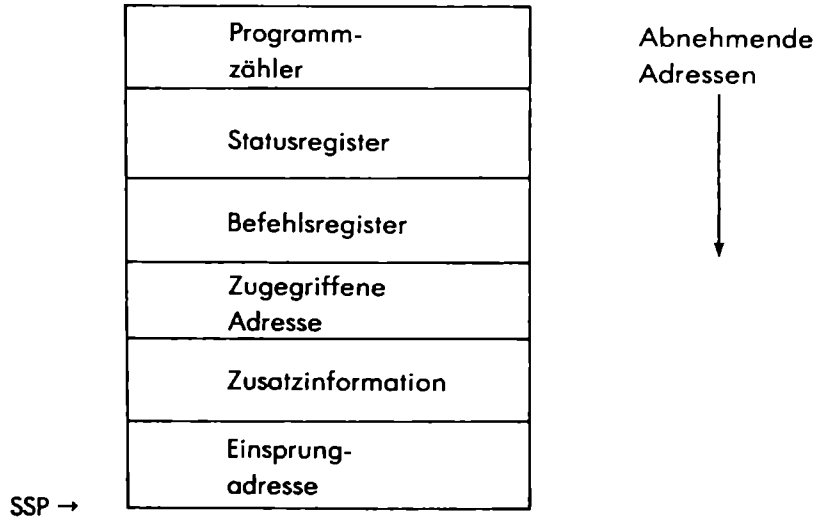
Die nun folgenden Label definieren die Adressen, die wir während der Initialisierung in die Vektortabelle eingetragen haben. Bei jedem Label wird einer der beiden Exception-Handler aufgerufen. Die bei diesem Sprung gerettete Rücksprungadresse kann dort als Index zur Feststellung der Fehlerursache dienen.

```
* Exceptions

B_EXCPT  BSR.S   EXCP1     Busfehler
A_EXCPT  BSR.S   EXCP1     Adreßfehler
I_EXCPT  BSR.S   EXCP2     Illegaler Befehl
D_EXCPT  BSR.S   EXCP2     Division durch Null
C_EXCPT  BSR.S   EXCP2     CHK
O_EXCPT  BSR.S   EXCP2     TRAPV
P_EXCPT  BSR.S   EXCP2     Privilegverletzung
T_EXCPT  BSR.S   EXCP2     TraceX_EXCPT  BSR.S   EXCP2   L1010
Y_EXCPT  BSR.S   EXCP2     L1111
S_EXCPT  BSR.S   EXCP2     Falscher Interrupt
```

```
* Unterbrechungen

INT       BSR.S     EXCP2       Unerwarteter Interrupt
INT7      BSR.S     EXCP2       Interrupt der Stufe 7

* Traps

TRP       BSR.S     EXCP2       Unerwarteter Trap
TRP14     BSR.S     EXCP2       Breakpoint
TRP15     BSR.S     EXCP2       Ende des Anwenderprogramms
```

Bei Auftreten eines Bus- oder Adreßfehlers ist die Behandlung der Exception aufwendiger als in den anderen Fällen. Aufgrund des Pre-Fetch-Mechanismus des 68000 ist der Programmzähler i.a. kleiner als es uns der auf dem Stack gerettete Wert glauben machen will. Auf dem Stack sind noch weitere Informationen abgelegt, darunter das Befehlsregister. Indem wir nun von der geretteten Adresse an abwärts im Speicher nach einem Wort suchen, das gleich diesem Wert ist, finden wir den tatsächlich im Augenblick des Fehlers ausgeführten Befehl.

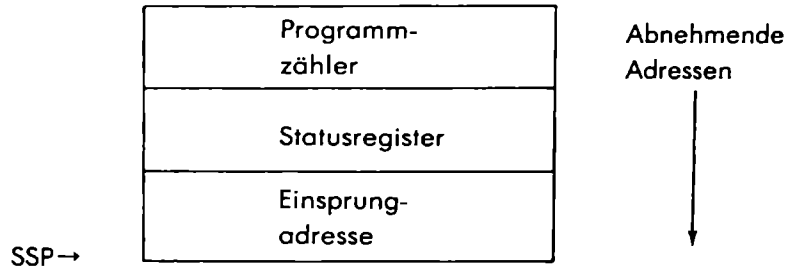
Auf dem Stack liegen nun die folgenden Informationen:




```
EXCP1     MOVEM.L   D0/A0,-(SP)     Register retten
          MOVE.L    22(SP),A0       Programmzähler vom Stack
          MOVE.W    18(SP),D0       Befehlsregister vom Stack
          CMP.W     -(A0),D0        A0 = A0 - 1; gefunden ?
          BEQ.S     EXCP10          Ja
          CMP.W     -(A0),D0        A0 = A0 - 1; gefunden ?
          BEQ.S     EXCP10          Ja
          CMP.W     -(A0),D0        A0 = A0 - 1; gefunden ?
          BEQ.S     EXCP10          Ja
          CMP.W     -(A0),D0        A0 = A0 - 1; gefunden ?
          BEQ.S     EXCP10          Ja
          SUBQ.L    #2,A0           Dann muß es diese Adresse sein
EXCP10    MOVE.L    A0,22(SP)       Korrigierten PC wieder auf dem
*                                   Stack ablegen
```

```
          MOVEM.L   (SP)+,D0/A0     Register wiederherstellen
          MOVE.L    (SP),8(SP)      Mit Rücksprungadresse
*                                   überschreiben
          ADDQ.L    #8,SP           SP korrigieren
*
* Übergang in den normalen Exception-Handler
*
```

Nun behandeln wir die einfachen Exceptions. Der Stack hat das folgende Aussehen:




```
EXCP2     BTRST     #SBIT,4(SP)         War das Programm im
*                                       Supervisormodus ?
          BNE.S     EXCP3               Ja → kein Anwenderprogramm
          MOVE.L    A0,-(SP)
          MOVEA.L   RAMBASE,A0          Zeiger auf Arbeitsspeicher
          LEA.L     RDUMPD(A0),A0       Save-Area für Register
          MOVEM.L   D0-D7/A0-A6,(A0)    Alle Register retten
          MOVE.L    (SP)+,D_A0(A0)      A0 vom Stack nach Save-Area
          MOVE.L    (SP)+,A1            Rücksprungadresse vom BSR
          MOVE.W    (SP)+,D_SR(A0)      Statusregister abspeichern
          BCLR      #TBIT,D_SR(A0)      Trace-Bit ausschalten
          MOVE.L    (SP)+,D_PC(A0)      Benutzer-PC abspeichern
          MOVE.L    USP,A2              USP holen
          MOVE.L    A2,D_A7(A0)         und als A7 speichern
```

Nun haben wir alle Register gerettet und können dazu übergehen, die im Code installierten Breakpoints durch die ursprünglichen Befehle zu ersetzen. Diese stehen in der Breakpoint-Tabelle in den hinteren beiden Bytes eines jeden Eintrags, auch wenn der Breakpoint nicht installiert ist.

```
          MOVEA.L   RAMBASE,A6      Anfang des Arbeitsspeichers
          LEA.L     BRKP(A6),A3     Zeiger auf die Breakpoint-Tabelle
          MOVEQ     #9,D0           Zähler initialisieren
BRKL      MOVE.L    (A3)+,A4        Adresse des Breakpoints
          MOVE.W    (A3)+,D1        Ursprünglicher Befehl
          CMPA.L    #0,A4           Breakpoint gesetzt ?
          BEQ.S     BRKL1           Nein → überspringen
          MOVE.W    D1,(A4)         Wiederherstellen
BRKL1     DBRA      D0,BRKL         Bis die Breakpoints ersetzt sind
          BRA.S     EXCP4           Exception weiterbearbeiten
```

War bei Auftreten der Exception das Supervisor-Bit im Statusregister gesetzt, so handelte es sich um einen im Monitor aufgetretenen Fehler (mit aller Wahrscheinlichkeit um einen Busfehler bei einem Speicherbefehl). Die Benutzer-Register werden daher nicht verändert, es wird aber die normale Meldung ausgegeben.

```
EXCP3    MOVE.L   (SP)+,A1        Rücksprungadresse von BSR
```

Nunmehr haben wir alle Benutzer-Register gerettet, und in A1 liegt die um zwei erhöhte Einsprungadresse. Um die richtige Adresse zu erhalten, müssen wir diesen Wert noch korrigieren.

Außerdem müssen wir den System-Stackzeiger auf den Wert zurücksetzen, den dieser beim Start des Monitors hatte. Nachdem wir dies alles durchgeführt haben, können wir wieder Interrupts zulassen, da wir auch bei einer Vielzahl unerwarteter Interrupts den Stackzeiger stets wieder auf den Anfang unseres Stackbereiches zurückgesetzt haben, bevor wir einen weitere Interrupt behandeln.

```
EXCP4    SUBQ.L   #2,A1            Zeiger auf den eigentlichen
*                                  Einsprung
         MOVE.L   I_RESET,SP       System-Stack rücksetzen
         MOVE.W   #INTSON,SR       Interrupts wieder zulassen
```

Nun prüfen wir, ob die Exception durch Tracen oder einen Breakpoint verursacht wurde.

```
         LEA.L    T_EXCPT,A0       Trace-Adresse
         CMPA.L   A0,A1            War's das?
         BEQ.S    EXCP5            Ja
         LEA.L    TRP14,A0         Breakpoint-Adresse
         CMPA.L   A0,A1            Aber das war's doch?
         BEQ.S    EXCP6            Ja
```

Sind wir bis hierher gelangt, so handelt es sich um irgendeine andere Ausnahmebedingung. Wir rufen dann das Unterprogramm WRABO auf, das eine dem Inhalt von A0 entsprechende Meldung ausgibt. Dieses Register zeigt immer noch auf den durch die Vektortabelle adressierten Einsprung in den Exception-Handler und kann somit einfach zur Auswahl der richtigen Meldung verwendet werden. Danach springen wir an den Anfang der Hauptschleife des Monitors zurück, um weitere Befehle einzulesen und auszuführen.

```
         BSR.S    WRABO            Meldung ausgeben
         BRA      ST1              Und wieder von vorne
```

Trat die Exception durch das gesetzte Trace-Bit auf, so sind zwei Fälle zu unterscheiden. Zum einen kann durch Eingabe von 'C' nach einem Breakpoint das Programm mit eingeschaltetem Trace-Bit wiederaufgenommen worden sein. Diesen Fall erkennen wir an dem gesetzten Flag BFLG. In diesem Fall nehmen wir einfach die Abarbeitung des Anwenderprogramms durch einen Sprung nach GO1 wieder auf. Dort wer-

den dann auch die Breakpoints (insbesondere der momentan nicht gesetzte) wieder installiert.

```
EXCP5    TST.B    BFLG(A6)        Letzter Befehl = 'C' ?
         BNE      GO1             ja → Weitermachen
```

Kommen wir nun zum normalen **Trace**. Zunächst geben wir mittels WRABO eine entsprechende Meldung und anschließend durch Aufruf von REGX die Belegungen aller Register aus.

Da normalerweise nicht nur ein Schritt im Tracemodus ablaufen soll, ändern wir die normale Befehlsabarbeitung derart, daß die Eingabe von Return wie die Eingabe von 'T' behandelt wird. Jeder andere Befehl wird normal ausgeführt. Dazu setzen wir A0 als Zeiger auf die Befehlstabelle und schreiben die Einsprungadresse in die Hauptschleife des Monitors auf den Stack. Dadurch erfolgt nach Abarbeitung eines Unterprogramms der Sprung wieder dorthin.

Um anzuzeigen, daß sich der Monitor in einem speziellen Modus befindet, geben wir den Buchstaben 't' als Prompt aus. Handelt es sich bei der Eingabe des Benutzers nicht um ein Return, so springen wir in das Unterprogramm SEARCH zwecks Bearbeitung der Eingabe an. Da wir die Adresse von ST1 auf dem Stack abgelegt haben, erfolgt der Rücksprung des von SEARCH aufgerufenen Unterprogramms dorthin.

Gab der Benutzer jedoch ein Return ein, so setzen wir das Trace-Bit und fahren mittels eines Sprungs nach GO1 mit der Abarbeitung des Benutzer-Programms fort.

```
         BSR.S    WRABO           Meldung ausgeben
         BSR      REGX            Register ebenfalls
         LEA.L    COMTAB,A0       Zeiger auf Befehlstabelle
         PEA.L    ST1             Rücksprungadresse zur
*                                 Befehlsschleife
         MOVE.B   #'t',D0         Neues Promptzeichen
         BSR      WRCH
         BSR      RDCH
         CMP.B    #CR,D0          Return eingegeben ?
         BNE      SEARCH          Nein → Normaler Befehl
         BSET     #TBIT,RDUMPSR(A6)  Trace-Bit im geretteten
*                                 Benutzer-Statusregister setzen
         BRA      GO1             Und weiter abarbeiten
```

Wir kommen jetzt zur Behandlung der Breakpoints. Indem wir einen Befehl ganz oder teilweise durch den Befehl TRAP #14 ersetzt haben, sind wir bei der Ausführung dieses Traps nunmehr bis hierhin gekommen. Der an der Stelle ursprünglich vorliegende Befehl wurde noch nicht ausgeführt. Als erstes müssen wir daher den Programmzähler um die zwei Byte verringern, die vom TRAP-Befehl benötigt wurden.

Danach geben wir mal wieder eine Meldung sowie die Inhalte der Register aus. Anschließend springen wir nach ST1, um den folgenden Befehl einzulesen. Soll das Benutzer-Programm fortgesetzt werden, so wird dieser Breakpoint nicht wieder installiert, da seine Adresse dem aktuellen Programmzähler entspricht. Bei Verwendung des C- oder T-Befehls erhält der Monitor die Kontrolle nach Abarbeitung eines Befehls wieder zurück; wir installieren dann vor dem nächsten Start des Benutzer-Programms den Breakpoint automatisch wieder.

```
EXCP6     SUBQ.L    #2,RDUMPPC(A6) PC korrigieren
          BSR.S     WRABO          Meldung ausgeben
          BSR       REGX           Und die Register
          BRA       ST1            Nächsten Befehl bearbeiten
```

Das letzte Unterprogramm unseres Monitors dient zur Ausgabe einer Meldung, wobei diese dem Inhalt von A1 entsprechend ausgewählt wird.

A1 zeigt auf das Label, das den Einsprungspunkt in den Exception-Handler markiert. Indem wir die Adresse des ersten möglichen Labels davon subtrahieren, erhalten wir einen Wort-Offset, da die einzelnen BSR.S-Befehle jeweils zwei Byte im Speicher belegen. Diesen Offset verwenden wir dann als Index für die Tabelle ABOTAB, welche die Offsets der Fehlermeldungen relativ zum Label ABOTAB enthält. Durch Addieren der Basisadresse dieser Tabelle erhalten wir die Adresse des durch WRITES auszugebenden Textes.

```
WRABO     LEA.L     B_EXCPT,A0     Erster Einsprung in den
*                                  Exception-Handler
          SUBA.L    A0,A1          Wort-Offset bestimmen
          LEA.L     ABOTAB,A2      Zeiger auf Tabelle der Offsets
          MOVE.L    A1,D0          D0 = Offset
          MOVE.W    0(A2,D0.L),A0  Offset des Textes in der Tabelle
          ADDA.L    A2,A0          A0 = Zeiger auf den Text
          BRA       WRITES         Ausgeben und Rücksprung
```

8.11 MELDUNGEN UND TABELLEN

Nun bleibt uns nur noch die Definition der im Monitor verwendeten Meldungen und Tabellen. Beginnen wir mit den Fehlermeldungen, die der Benutzer bei Eingabe eines unbekannten Befehls erhält, sowie unserer Begrüßungsmeldung.

```
MESS1     DC.B      'MC68000 Monitor V1.2',0
MESS2     DC.B      'Unbekannter Befehl',0
MESS3     DC.B      'Hexadezimalzahl erwartet',0
MESS4     DC.B      'Ungültiger Speicherbefehl',0
MESS5     DC.B      'Adresse ist ungerade',0
MESS6     DC.B      'Falsche Registernummer',0
MESS7     DC.B      'Supervisor-Bit ist gesetzt',0
MESS8     DC.B      'Falsche Breakpoint-Nummer',0
```

Es folgen die Meldungen für Exceptions, unerwartete Interrupts und Traps.

```
AB1     DC.B    'Busfehler',0
AB2     DC.B    'Adreßfehler',0
AB1     DC.B    'Illegaler Befehl',0
AB4     DC.B    'Division durch Null',0
AB5     DC.B    'CHK',0
AB6     DC.B    'TRAPV',0
AB7     DC.B    'Privilegverletzung',0
AB8     DC.B    'Trace ...',0
AB9     DC.B    'Illegaler Befehl (1010)',0
AB10    DC.B    'Illegaler Befehl (1111)',0
AB11    DC.B    'Unechte Interrupt',0
AB12    DC.B    'Unerwartete Interrupt ',0
AB13    DC.B    'Interrupt der Stufe 7',0
AB14    DC.B    'TRAP',0
AB15    DC.B    'Breakpoint',0
AB16    DC.B    'Ende des Benutzer-Programms',0
```

Die Adressen dieser Meldungen sind in der nun folgenden Tabelle abgelegt. Sie werden als Offsets relativ zum Anfang dieser Tabelle definiert, um die Positionsunabhängigkeit zu erhalten. Die Anordnung in dieser Tabelle entspricht der der Label am Anfang des Exception-Handlers.

```
ABOTAB  DC.W    (AB1-ABOTAB)
        DC.W    (AB2-ABOTAB)
        DC.W    (AB3-ABOTAB)
        DC.W    (AB4-ABOTAB)
        DC.W    (AB5-ABOTAB)
        DC.W    (AB6-ABOTAB)
        DC.W    (AB7-ABOTAB)
        DC.W    (AB8-ABOTAB)
        DC.W    (AB9-ABOTAB)
        DC.W    (AB10-ABOTAB)
        DC.W    (AB11-ABOTAB)
        DC.W    (AB12-ABOTAB)
        DC.W    (AB13-ABOTAB)
        DC.W    (AB14-ABOTAB)
        DC.W    (AB15-ABOTAB)
        DC.W    (AB16-ABOTAB)
```

Die nun folgenden Tabellen sind zum Gebrauch für das Unterprogramm SEARCH bestimmt, d.h. daß (außer beim letzten) bei allen Einträgen das erste Byte den Wert Null hat. Das zweite Byte enthält einen Kleinbuchstaben und die beiden letzten die Adresse eines Unterprogramms. Zur Erhaltung der Positionsunabhängigkeit werden diese auch hier als Offset relativ zum Tabellenanfang definiert.

Die erste Tabelle enthält die normalen Monitorbefehle.

```
COMTAB  DC.W    CR              Nur ein Return
        DC.W    (SKIPNL-COMTAB)
        DC.W    'm'             Speicherbefehle
        DC.W    (MEM-COMTAB)
```

```
        DC.W    'r'             Ausgabe der Registerinhalte
        DC.W    (REG-COMTAB)
        DC.W    'd'             Datenregister verändern
        DC.W    (SETD-COMTAB)
        DC.W    'a'             Adreßregister verändern
        DC.W    (SETA-COMTAB)
        DC.W    'p'             PC verändern
        DC.W    (SETP-COMTAB)
        DC.W    's'             SR verändern
        DC.W    (SETS-COMTAB)
        DC.W    'g'             Benutzerprogramm starten
        DC.W    (GO-COMTAB)
        DC.W    't'             Benutzerprogramm tracen
        DC.W    (TGO-COMTAB)
        DC.W    'b'             Breakpoint
        DC.W    (BRK-COMTAB)
        DC.W    'c'             Fortsetzung nach Breakpoint
        DC.W    (CONT-COMTAB)
        DC.W    $FF00           Ende der Tabelle
        DC.W    (COMERR-COMTAB)
```

Zum Abschluß nun noch die Tabelle mit den Speicherbefehlen.

```
MEMTAB  DC.W    CR              Nächste Speicherstelle öffnen
        DC.W    (NMEM-MEMTAB)
        DC.W    '^'             Vorherige Speicherstelle öffnen
        DC.W    (PMEM-MEMTAB)
        DC.W    '='             Nochmals ausgeben
        DC.W    (SKIPNL-MEMTAB)
        DC.W    'b'             Größe auf Byte setze
        DC.W    (SETB-MEMTAB)
        DC.W    'w'             Größe auf Wort setzen
        DC.W    (SETW-MEMTAB)
        DC.W    'l'             Größe auf Vollwort setzen
        DC.W    (SETL-MEMTAB)
        DC.W    $FF00           Ende der Tabelle, Fehler bei den
        DC.W    (MEMERR-MEMTAB) Speicherbefehlen
        END                     e des Monitors
```

Anhang A
Der Befehlssatz des 68000

Der Zustand der Condition-Codes im Statusregister nach Ausführung eines Befehls wird folgendermaßen angezeigt.

–	Unbeeinflußt
0	Gelöscht
1	Gesetzt
A	Verändert, falls der Operand kein Adreßregister ist
C	Dem Resultat entsprechend verändert
P	Eventuell dem Resultat entsprechend verändert
U	Undefiniert

Die verschiedenen Adressierungsarten werden gemäß der folgenden Tabelle angeben.

An	Ein beliebiges Adreßregister
Dn	Ein beliebiges Datenregister
Rn	Ein beliebiges Register
(An)	Indirekt durch ein Adreßregister
d(An)	Indirekt durch ein Adreßregister mit Displacement
–(An)	Indirekt durch ein Adreßregister mit Predecrement
(An)+	Indirekt durch ein Adreßregister mit Postincrement
<ea>	Beliebige Adressierungsart
<aea>	Adressierungsart vom Typ 'alterable'
<cea>	Adressierungsart vom Typ 'control'
<dea>	Adressierungsart vom Typ 'data'
<caea>	Adressierungsart vom Typ 'control alterable'
<daea>	Adressierungsart vom Typ 'data alterable'
<maea>	Adressierungsart vom Typ 'memory alterable'
<rl>	Registerliste
<imm>	Direktoperand

Die verschiedenen Kategorien der Operandenadressen finden Sie in der folgenden Tabelle.

Adressierungsart	*data*	*memory*	*control*	*alterable*
Dn	x	–	–	x
An	–	–	–	x
relativ zum PC	x	x	x	–
rel. + Index	x	x	x	–
(An)	x	x	x	x
d16(An)	x	x	x	x
d8(An,Ri)	x	x	x	x
–(An)	x	x	–	x
(An)+	x	x	–	x
absolut	x	x	x	x
Direktoperand	x	x	–	–

Die möglichen Längenformate werden mit B für Byte, W für Wort und L für Vollwort (long word) verschlüsselt. Dann folgen die Zustände der Condition-Codes nach der Ausführung des Befehls, und anschließend steht bei jedem Befehl ein Verweis auf die Seite im Text, wo er genauer erklärt wird. Abschließend folgen einige Beispiele für die verschiedenen möglichen Adressierungsmodi. Haben mehrere Befehle die gleiche Form, so wird in diesen Beispielen nur der erste verwendet. Unterscheidet sich die Syntax bei verschiedenen Befehlen, so wird jede mögliche Variation aufgeführt.

Name	*Beschreibung*	*Größe*	*N Z V C X*	*Seite*
ABCD	Dezimale Addition Beispiele: ABCD Dn,Dn ABCD −(An),−(An)	B	U P U C C	104

ADD
ADDA
ADDI
ADDQ
ADDX
	Binäre Addition
	
	Beispiele:
	ADD <ea>,Dn
	ADD Dn,<maea>
	ADDA <ea>,An
	ADDI # <imm>,<daea>
	ADDQ # <imm>,<aea>
	ADDX Dn,Dn
	ADDX −(An),−(An)
		BWL
		WL
		BWL
		BWL
		BWL
			C C C C C
			− − − − −
			C C C C C
			A A A A A
			C P C C C
				91
				92
				92
				92
				92

AND ANDI ANDI mit CCR ANDI mit SR	Logisches Und (privilegiert) Beispiele: AND <dea>,Dn AND Dn,<maea> ANDI # <imm>,<daea> ANDI #<imm>,CCR ANDI #<imm>,SR	BWL BWL B W	C C 0 0 − C C 0 0 − P P P P P P P P P P	112 113 113 113

Name	*Beschreibung*	*Größe*	*N Z V C X*	*Seite*
ASL ASR	Arithm. Shift links Arithm. Shift rechts Beispiele: ASL Dn,Dn ASL # <imm>,Dn ASL <maea>	BWL BWL	C C C C C C C C C C	114 114
Bcc BRA BSR	Bedingter Sprung Unbedingter Sprung Sprung zum Unterprogramm Beispiel: Bcc <label>	BW BW BW	— — — — —	57 58 75
BCHG BCLR BSET BTST	Bit testen und ändern Bit testen und löschen Bit testen und setzen Bit testen Beispiele: BCHG Dn,<daea> BCHG # <imm>,<daea> BTST Dn,<dea> BTST # <imm>,<dea>	B L B L B L B L	— C — — — — C — — — — C — — — — C — — —	116 116 116 116
CHK	Test, evtl. Trap Beispiel: CHK <dea>,Dn	W	U U U U —	132
CLR	Löschen Beispiel: CLR <daea>,Dn	BWL	0 1 0 0 —	60

CMP
CMPA
CMPI
CMPM
	Vergleich
	
	Beispiele:
	CMP <ea>,Dn
	CMPA <ea>,An
	CMPI # <imm>,<daea>
	CMPM (An)+,(An)+
		BWL
		WL
		BWL
		BWL
			C C C C —
			C C C C —
			C C C C —
			C C C C —
				56
				56
				56
				56

Name	*Beschreibung*	*Größe*	*N Z V C X*	*Seite*
DBcc	Dekrement und bedingter Sprung	W	– – – – –	63
DBRA	Dekrement und Sprung	W	– – – – –	63
	Beispiel: DBcc Dn, <label>			
DIVS	Division (signiert)	W	C C C 0 –	100
DIVU	Division (unsigniert)	W	C C C 0 –	100
	Beispiel: DIVS <dea>,Dn			
EOR	Exklusives Oder	BWL	C C 0 0 –	112
EORI		BWL	C C 0 0 –	113
EORI	mit CCR	B	P P P P P	113
EORI	mit SR (privilegiert)	W	P P P P P	113
	Beispiele: EOR Dn, <daea> EORI # <imm>, <daea> EORI # <imm>,CCR EORI # <imm>,SR			
EXG	Registeraustausch	L	– – – – –	95
	Beispiel: EXG Rn,Rn			
EXT	Sign-Extension	WL	C C 0 0 –	95
	Beispiel: EXT Dn			
JMP	Sprung	–	– – – – –	80
JSR	Sprung ins Unterprogramm	–	– – – – –	82
	Beispiel: JMP <cea>			
LEA	Operandenadresse laden	L	– – – – –	83
	Beispiel: LEA <cea>,An			
LINK	Binden eines Unterprogramms		– – – – –	87

Name	*Beschreibung*	*Größe*	*N Z V C X*	*Seite*
LSL	Logischer Shift links	BWL	C C 0 C C	113
LSR	Logischer Shift rechts	BWL	C C 0 C C	114
	Beispiele: LSL Dn,Dn LSL # <imm>,Dn LSL <maea>			
MOVE	Datentransport	BWL	C C 0 0 —	51
MOVEA		WL	— — — — —	53
MOVEM	Registertransport	WL	— — — — —	73
MOVEP	Transport zu peripheren Geräten	WL	— — — — —	68
MOVEQ		L	C C 0 0 —	61
MOVE n. CCR		W	C C C C C	128
MOVE n. SR	(privilegiert)	W	C C C C C	128
MOVE von SR		W	— — — — —	128
MOVE mit USP	(privilegiert)	L	— — — — —	126

	Beispiele:
	MOVE <ea>,<daea>
	MOVEA <ea>,An
	MOVEM <rl>,−(An)
	MOVEM <rl>,<cea>
	MOVEM (An)+,<rl>
	MOVEM <cea>,<rl>
	MOVEP Dn,d(An)
	MOVEP d(An),Dn
	MOVEQ # <imm>,Dn
	MOVE <dea>,CCR
	MOVE <dea>,SR
	MOVE SR,<daea>
	MOVE USP,An
	MOVE An,USP			

MULS	Multiplikation (signiert)	W	C C C 0 —	94
MULU	Multiplikation (unsigniert)	W	C C C 0 —	94
	Beispiel: MULS <dea>,Dn			
NBCD	Dezimale Negation	B	U P U C C	104
	Beispiel: NBCD <daea>			

Name	*Beschreibung*	*Größe*	*N Z V C X*	*Seite*
NEG	Binäre Negation	BWL	C C C C C	94
NEGX		BWL	C P C C C	94
	Beispiel: NEG <daea>			
NOP	Keine Operation	—	– – – – –	31
	Beispiel NOP			
NOT	Logisches Komplement	BWL	C C 0 0 –	111
	Beispiel: NOT <daea>			
OR	Logisches Oder	BWL	C C 0 0 –	111
ORI		BWL	C C 0 0 –	113
ORI mit CCR		B	P P P P P	113
ORI mit SR	(privilegiert)	W	P P P P P	113
	Beispiel: OR <dea>,Dn OR Dn,<maea> ORI # <imm>,<daea> ORI <imm>,CCR ORI # <imm>,SR			
PEA	Operandenadresse auf dem Stack ablegen	L	– – – – –	84
	Beispiel: PEA <cea>			
RESET	Rücksetzen (privilegiert)	–	– – – – –	133
	Beispiel: RESET			
ROL	Rotation links	BWL	C C 0 C –	115
ROXL		BWL	C C 0 C C	115
ROR	Rotation rechts	BWL	C C 0 C –	115
ROXR		BWL	C C 0 C C	115
	Beispiel: ROL Dn,Dn ROL # <imm>,Dn ROL <maea>			

Name	*Beschreibung*	*Größe*	*N Z V C X*	*Seite*
RTE	Rücksprung von Exception	—	C C C C C	127
RTR	Rücksprung und CCR setzen	—	C C C C C	128
RTS	Rücksprung aus Unterprogramm	—	— — — — —	76
	Beispiel: RTE			
SBCD	Dezimales Subtrahieren	B	U P U C C	104
	Beispiele: SBCD Dn,Dn SBCD −(An),−(An)			
Scc	Bedingtes Setzen	B	— — — — —	62
	Beispiel: Scc <daea>			
STOP	Prozessor anhalten (privilegiert)	—	C C C C C	133
	Beispiel: STOP # <imm>			
SUB	Binäres Subtrahieren	BWL	C C C C C	93
SUBA		WL	— — — — —	93
SUBI		BWL	C C C C C	93
SUBQ		BWL	C C C C C	93
SUBX		BWL	C P C C C	93

	Beispiele:
	SUB <ea>,Dn
	SUB Dn,<maea>
	SUBA <ea>,An
	SUBI # <imm>,<daea>
	SUBQ # <imm>,<aea>
	SUBX Dn,Dn
	SUBX −(An),−(An)			

SWAP	Registerhälften tauschen	W	C C 0 0 —	95
	Beispiel: SWAP Dn			
TAS	Bit testen und setzen	B	C C 0 0 —	117
	Beispiel: TAS <daea>			

TRAP TRAPV	Trap-Exception erzeugen Trap bei Überlauf Beispiele: TRAP # <imm> TRAPV	— —	– – – – – – – – – – – –	131 132
TST	Mit Null vergleichen Beispiel: TST <daea>	BWL	C C 0 0 –	60
UNLK	Entbinden von Unterprogrammen	—	– – – – –	87

Test von Bedingungen

In der folgenden Tabelle finden Sie die möglichen Kombinationen von Bedingungen, die in Bcc, DBcc und Scc spezifiziert werden können. Zusätzlich können bei den Befehlen DBcc und Scc auch T (true = wahr) und F (false = falsch) angegeben werden. Einige Assembler akzeptieren zusätzlich auch DBRA statt DBF und die Tests HS und LO statt CC und CS.

In der Tabelle bedeutet C, daß das C-Bit gesetzt sein muß, damit die Bedingung erfüllt ist, und C', daß es gelöscht sein muß. Mit ' | ' verknüpfte Bedingungen müssen beide wahr sein, wogegen von den durch < > / verbundenen nur eine erfüllt sein muß.

Name	*Bedingung*	*Test*
CC	Kein Übertrag	C'
CS	Übertrag	C
EQ	Gleichheit	Z
NE	Ungleichheit	Z'
PL	Positiv	N'
MI	Negativ	N
VC	Kein Überlauf	V'
VS	Überlauf	V
HI	Logisches >	C' & Z'
LS	Logisches <=	C | Z
HS	Logisches >=	C'
LO	Logisches <	C
GT	>	(N & V & Z') | (N' & V' & Z')
GE	>=	(N & V) | (N' & V')
LE	<=	Z | (N & V') | (N' & V)
LT	<	(N & V') | (N' & V)

Anhang B

Beispielprogramme für Sinclair QL, Atari 520ST und Commodore Amiga

Die bisher in diesem Buch aufgeführten Beispiele waren absichtlich so entworfen, daß sie sich auf keinen bestimmten Computer und kein bestimmtes Betriebssystem beziehen. Oft ist es jedoch nützlich, ein echtes Beispielprogramm zu sehen, das sofort in einen konkreten Computer eingegeben werden kann. Im folgenden wollen wir einige derartige Beispiele vorstellen.

Das Beispielprogramm ist allerdings recht einfach. Der Benutzer wird durch einen Prompt auf dem Bildschirm um die Eingabe seines Namens gebeten, und seine Antwort wird eingelesen. Anschließend wird eine weitere Meldung, zusammen mit seinem Namen, ausgegeben.

Wir werden drei verschiedene Beispiele für drei verschiedene **Mikrocomputer** vorstellen, die auf dem 68000 basieren – dem **Sinclair QL**, dem **Atari 520ST** und Commodores **Amiga**. Diese drei Computer verwenden verschiedene Betriebssyteme, und es ist interessant, die Ähnlichkeiten und Unterschiede dieser Systeme zu untersuchen.

Sinclair QL

Das erste Beispiel ist für den QL, der unter dem Betriebssystem **QDOS** läuft. Alle Systemaufrufe werden durch TRAP-Befehle erzeugt, wobei nur TRAP #1, #2 und #3 verwendet werden. Die Nummer der gewünschten Funktion wird vorher im Register D0 abgelegt, andere eventuell benötigte **Parameter** werden in andere Register geladen. Wir definieren zunächst einige symbolische Namen für die Codes der Funktionen, die einen Job beenden, einen Kanal öffnen bzw. schließen, eine Zeile einlesen bzw. ausgeben.

```
MT_FRJOB EQU     $05
IO_OPEN  EQU     $01
IO_CLOSE EQU     $02
IO_FLINE EQU     $02
IO_SSTRG EQU     $07
```

Normalerweise ist es nützlich, für bestimmte Funktionen (wie etwa Systemaufrufe) Makros zu definieren. Ein **Makro** wird bei Aufruf zu dem gewünschten Code expandiert und spart dadurch Tipparbeit und erhöht die Lesbarkeit des Programms.

Das im folgenden definierte Makro erhält zwei Parameter; der erste gibt die Codenummer der Funktion an und wird in D0 abgelegt; der zweite gibt an, welcher TRAP-Befehl auszuführen ist.

```
QDOS    MACRO
        MOVEQ     #\1,D0
        TRAP      #\2
        ENDM
```

Nun können wir mit unserem einfachen Programm beginnen.

Zunächst müssen wir einen Ein-/Ausgabekanal für die **Konsole** öffnen, damit wir auf den Bildschirm schreiben und von der Tastatur lesen können. Der QL unterstützt eine einfache **Fenstertechnik**, so daß wir einen Kanal für ein Standard-Fenster öffnen können, indem wir 'CON_' als Namen des Geräts angeben, das wir öffnen wollen. Im Register A0 legen wir einen Zeiger auf den Namen des Geräts ab, in D3 geben wir den öffnungsmodus an. Außerdem müssen wir in D1 eine Job-Bezeichnung spezifizieren, die für den laufenden Job den Wert −1 hat. Der Bezeichner des geöffneten Kanals wird im Register A0 zurückgegeben.

```
        MOVEQ     -1,D1          Laufender Job
        MOVEQ     #2,D3          Modus: exklusiv
        LEA.L     DEVNAME,A0     Zeiger auf den Gerätenamen
        QDOS      IO_OPEN,2      Kanal öffnen
```

Jetzt können wir in dem geöffneten Fenster einen **Prompt** ausgeben. Dazu verwenden wir die QDOS-Funktion zur Ausgabe eines Textes. Diese benötigt in D2 die Länge und in A1 einen Zeiger auf den Anfang des Textes. Außerdem müssen wir im Register D3 ein **Zeitlimit** für die Operation angeben; in den meisten Fällen wird dafür −1 verwendet, wodurch das Zeitlimit auf unendlich gesetzt wird. Der Kanalbezeichner, den wir beim öffnen in A0 zurückerhalten haben, wird genau dort benötigt; er wird durch den folgenden Aufruf nicht zerstört.

```
        MOVEQ     #27,D2         Länges des Textes
        MOVEQ     #-1,D3         Kein Zeitlimit
        LEA.L     PROMPT,A1      Zeiger auf den Text
        QDOS      IO_SSTRG,3     Text ausgeben
```

Nun müssen wir die Antwort von der Tastatur einlesen. Der QDOS-Aufruf zum Einlesen einer Zeile sammelt die eingegebenen Zeichen in einem spezifizierten **Puffer**; er sorgt auch für das **Echoen** der Zeichen und die Behandlung der **Löschtaste**. Dieses Unterprogramm terminiert erst, wenn der Benutzer ein RETURN eingibt oder der Puffer voll ist. Wir werden das erste Wort des Puffers verwenden, um die Länge des eingegebenen Textes zu speichern, die in D1 zurückgegeben wird. Daher setzen wir A3 auf die Anfangsadresse des Puffers; A1 zeigt zwei Byte weiter auf den Bereich für die Zeichen selbst. Die Länge des Puffers wird in D2 abgelegt; das Zeitlimit und der Kanal-Bezeichner stehen noch vom letzten Systemaufruf unverändert in D3 und A0. Nach Rückkehr von dem Systemaufruf speichern wir die Länge der eingelesenen Zeichenkette in dem freigelassenen Wort des Puffers ab, auf das A3 zeigt.

```
LEA.L     PUFFER,A3      Zeiger auf den Puffer
LEA.L     2(A3),A1        Erstes Wort überspringen
MOVEQ     #30,D2         Länge des Puffers
QDOS      IO_FLINE,3     Eingabezeile lesen
MOVE.W    D1,(A3)        Länge der Eingabe abspeichern
```

Jetzt wollen wir den nächsten Text ausgeben, wobei wir denselben Mechanismus wie für den Prompt verwenden. Anschließend soll noch der Name des Benutzers so ausgegeben werden, wie er eingegeben wurde. Dazu setzen wir D2 auf die Länge des Textes, die in dem durch A3 adressierten Wort steht. Wir holen uns daher diesen Wert und setzen A1 so, daß es auf den eigentlichen Pufferanfang zeigt.

```
MOVEQ     #6,D2          Länge des Textes
MOVEQ     #-1,D3         Kein Zeitlimit
LEA.L     MESS,A1        Zeiger auf die Meldung
QDOS      IO_SSTRG,3     Meldung ausgeben
MOVEQ     #0,D2          D2 löschen
MOVE.W    (A3),D2        Länge des Namens
LEA.L     2(A3),A1       Zeiger auf den Namen
QDOS      IO_SSTRG,3     Namen ausgeben
```

Wir haben nun unser Beispiel fast vollendet; wir müssen nur noch das Fenster wieder schließen und den laufenden Job beenden. Der Aufruf zum Schließen benötigt nur den Kanal-Bezeichner, der nach wie vor in A0 steht. Der anschließende Aufruf zum Beenden des Jobs benötigt in D1 den Job-Bezeichner, hier also −1 für den laufenden Job. In D3 wird vorher der **Return-Code** Null eingetragen.

```
QDOS      IO_CLOSE,2     Kanal schließen
MOVEQ     #-1,D1         Laufender Job
MOVEQ     #0,D3          Return-Code 0
GDOS      MT_FRJOB,1     Job Beenden
```

Es gibt unter QDOS zwei Möglichkeiten, Assemblerprogramme ablaufen zu lassen. Die erste besteht darin, sie durch EXEC aufzurufen, wobei ein neuer Job generiert wird, der durch den obigen Aufruf beendet wird. Es ist aber auch möglich, mittels CALL ein Programm von einem laufenden Job aus aufzurufen, der durch den obigen Systemaufruf nicht beendet werden kann. In deisem Fall wird eine entsprechenden Rücksprungadresse auf dem Stack abgelegt, so daß wir nach dem Setzen des Return-Codes mit einem einfachen RTS-Befehl zurückspringen können.

```
MOVEQ     #0.D0      Return-Code setzen
RTS
```

Der Datenbereich enthält den Platz für den Puffer und den Namen des Geräts, das wir öffnen wollen. Vor diesem Namen steht ein Wort, daß die Länge des Namens enthält. Zum Schluß kommen noch die beiden verwendeten Texte.

```
PUFFER    DS.W       1
          DS.B       30
DEVNAME   DC.W       4
          DC.B       'CON_'
MESS      DC.B       'Hallo '
PROMPT    DC.B       'Geben Sie Ihren Namen ein: '
          END
```

Atari 520ST

Die Version für den **520ST** ähnelt der für den QL. Der ST läuft unter dem Betriebssystem **TOS**, bei dem Systemaufrufe ebenfalls durch TRAP-Befehle realisiert werden. Hierbei werden allerdings alle Aufrufe über TRAP #1 durchgeführt, und alle zu übergebenden Parameter werden — den **Aufrufkonventionen** von C entsprechend —auf dem Stack abgelegt. Das erste Argument (das zuletzt auf dem Stack abgelegte) ist der Funktionscode, der den Typ des gewünschten Aufrufs spezifiziert. Wir werden die Funktionscodes zum Beenden eines Programms, zur Ausgabe eines mit Null abgeschlossenen Textes und zum Einlesen eines Textes von der Tastatur benötigen.

```
TERM    EQU     $0      Laufendes Programm beenden
WRST    EQU     $9      Einen Text ausgeben
RDST    EQU     $A      Einen Text einlesen
```

Nun definieren wir ein Makro, um den Funktionscode auf dem Stack abzulegen und den TRAP-Befehl zum Aufruf einer Systemroutine durchzuführen. Beachten Sie, daß durch das Makro keiner der Parameter auf dem Stack abgelegt und dieser anschließend auch nicht wiederhergestellt wird.

```
CALL    MACRO
        MOVE.W  #\1,-(SP)
        TRAP    #1
        ENDM
```

Die von uns verwendeten Systemaufrufe beziehen sich auf das Lesen von der Standard-Eingabe (normalerweise Tastatur) und das Schreiben auf die Standard-Ausgabe (normalerweise Bildschrim). Dafür benötigen wir keinerlei Initialisierung und können uns somit sofort der Ausgabe der Prompts widmen. Dazu müssen wir mittels PEA die Adresse des durch Null begrenzten Textes auf dem Stack ablegen. Dann geben wir den Text aus, indem wir mit unserem Makro die entsprechende Systemfunktion aufrufen. Dabei dürfen wir nicht vergessen, anschließend den Stack wiederherzustellen. Wir können dies durch einen ADDQ-Befehl erreichen, sollten aber erwähnen, daß es besser ist, den Befehl LEA n(SP),SP anstatt der langen Form von ADD zu verwenden, wenn der Stack um mehr als acht Byte vergrößert wurde.

```
        PEA.L   PROMPT      Zeiger auf den Text
        CALL    WRST        Ausgeben
        ADDQ.L  #6,SP       Stack wiederherstellen
```

Als nächstes müssen wir die Antwort des Benutzers einlesen. Wir verwenden dafür den Systemaufruf zum Lesen von der Konsole, der die Adresse eines Puffers benötigt. Das erste Byte dieses Puffers enthält seine Länge, in das zweite Byte wird durch den Aufruf die Anzahl der eingelesenen Zeichen eingetragen, die sich dann ab dem dritten Byte im Puffer finden. Hinter dem letzten eingelesenen Zeichen findet sich als Abschluß eine Null. Der Systemaufruf terminiert, wenn der Benutzer ein RETURN eingibt oder der Puffer voll ist. Wie schon beim QL behandelt das Betriebssystem Details wie das Echoen der Zeichen und die Behandlung der Löschtaste.

```
PEA.L     PUFFER     Adresse des Puffers
CALL      RDST       Text einlesen
ADDQ.L    #6,SP      Stack wiederherstellen
```

Nun müssen wir die Meldung ausgeben, die vor dem Namen des Benutzers stehen soll. Wir verwenden dafür den gleichen Mechanismus wie bei der Ausgabe des Prompts.

```
PEA.L     MESS       Adresse des Textes
CALL      WRST       Text ausgeben
ADDQ.L    #6,SP      Stack wiederherstellen
```

Als nächstes folgt die Ausgabe des Textes, den der Benutzer eingegeben hat. Dabei verwenden wir nur den Teil des Puffers, der diesen Text enthält, und nicht wie vorher den ganzen Puffer, der noch seine Länge und die Anzahl der eingegebenen Zeichen enthält. Die Null zum Abschluß des Textes hat das Betriebssystem dankenswerterweise bereits eingetragen.

```
PEA.L     ZEICHEN    Adresse des Textes
CALL      WRST       Text ausgeben
ADDQ.L    #6,SP      Stack wiederherstellen
```

Zum Schluß müssen wir das Programm noch beenden, was sich durch einen einfachen Aufruf des Betriebssystems bewerkstelligen läßt.

```
CALL      TERM       Ende des Programms
```

Der Datenbereich enthält den notwendigen Platz für den Puffer inklusive der Länge des Puffers und der Anzahl der eingelesenen Zeichen in den ersten beiden Bytes. Wir verwenden hier ein zweites Label, um auf den tatsächlichen Text verweisen zu können. Beachten Sie, daß wir dem zweiten Text einen **Zeilenvorschub** ($A) voranstellen, da das Betriebssystem nur einen **Wagenrücklauf** echot, wenn der Benutzer die RETURN-Taste betätigt. Vergessen Sie nicht, daß alle Texte durch eine Null abgeschlossen werden müssen.

```
PUFFER    DC.B      30
          DS.B      1
ZEICHEN   DS.B      30
MESS      DC.B      $A,'Hallo ',0
PROMPT    DC.B      'Geben Sie Ihren Namen ein: ' , 0
          END
```

Commodore Amiga

Unser letztes Beispiel ist für den **Amiga**. Dessen Betriebssystem verwendet — anders als die der beiden anderen Computer — keine TRAP-Befehle für Systemaufrufe; stattdessen werden verschiedene Teile des Betriebssystems als Bibliotheken zur Verfügung gestellt, die in Wirklichkeit nur Sprungtabellen enthalten. Der Aufruf einer Funktion des **AmigaDOS** geschieht, indem ein bestimmter Offset vom Anfangszeiger der

entsprechenden Bibliothek angesprungen wird. Wir benötigen nur zwei Bibliotheken, die mit 'Exec' und 'AmigaDOS' bezeichnet werden. Die Exec-Bibliothek ist die Grundbibliothek des Systems; ihr Anfangszeiger findet sich in der Speicherstelle 4. Zum Verweis auf diese Stelle definieren wir einen Namen.

```
SYSBASIS EQU     $4      Anfangszeiger der Exec-Bibliothek
```

Um eine Exec- oder AmigaDOS-Funktion aufzurufen, müssen wir mittels JSR einen bestimmten Offset von dem in A6 vorliegenden Anfangszeiger anspringen. Die tatsächlichen Werte dieser Offsets brauchen wir nicht zu kennen, da sie als externe Definitionen in der System-Bibliothek 'amiga.lib' vorliegen. Wenn wir den **Linker** auffordern, diese Bibliothek zu untersuchen, können wir den Namen der gewünschten Funktion mit Hilfe der XREF-Anweisung als extern definieren; der Linker trägt dann den tatsächlichen Wert ein. Wir müssen dabei dem gewünschten Namen das Präfix **LVO** (für 'Library Vector Offset', Offset vom Bibliotheksvektor) voransetzen, um den Bibliotheksoffset von den Einsprungstellen der C-Funktionen zu unterscheiden, die ebenfalls in dieser Bibliothek stehen.

```
CALL     MACRO
         XREF     _LVO\1
         JSR      _LVO\1(A6)
         ENDM
```

Als erstes müssen wir die Funktion 'OpenLibrary' (öffne Bibliothek) von Exec aufrufen, um den Anfangszeiger für die AmigaDOS-Bibliothek zu erhalten. Wir erreichen dies, indem wir A6 mit dem Anfangszeiger von Exec laden und in A1 einen Zeiger auf den Namen der gewünschten Bibliothek ablegen. Zusätzlich müssen wir in D0 eine **Versionsnummer** angeben, durch die die kleinste akzeptable Versionsnummer des Systems spezifiziert wird, unter der das Programm laufen kann. Da unser Programm mit jeder Version des Betriebssystems läuft, verwenden wir hier die Versionsnummer Null. Nach vollendetem Aufruf retten wir für die folgenden Aufrufe von AmigaDOS den Anfangszeiger der AmigaDOS-Bibliothek in A6.

```
         MOVEA.L  SYSBASIS,A6   Zeiger für Exec-Bibliothek
         LEA.L    DOSNAME,A1    Zeiger auf Namen der
*                               DOS-Bibliothek
         MOVEQ    #0,D0         Version Null oder jünger
         CALL     OpenLibrary   DOS-Bibliothek öffnen
         TST.L    D0            Trat ein Fehler auf?
         BEQ.S    FEHLER        Ja ...
         MOVEA.L  D0,A6         Anfangszeiger retten
```

Nun müssen wir die Kanäle ausfindig machen, durch die die Standard-Ein-/Ausgabe repräsentiert wird. Wurde das Programm von der **Kommandozeile** (CLI) her aufgerufen, so liefern die Aufrufe von 'Input' und 'Output' die entsprechenden **File-Handles.** Wurde das Programm von der **Workbench** her aufgerufen, so müßten wir ein Fenster eröffnen; dies ist aber zu kompliziert, um hier behandelt werden zu können. Die File-Handles legen wir in D7 und D6 ab.

```
CALL    Input
MOVE.L  D0,D7     D7 enthält den File-Handle für stdin
CALL    Output
MOVE.L  D0,D6     D6 enthält den File-Handle für stdout
```

Jetzt laden wir die Adresse des Prompts nach A0 und springen in ein kleines Unterprogramm, das diesen Text ausgibt. Dabei halten wir uns an die Konvention, daß die Länge des Textes im seinem ersten Byte steht, das selbstverständlich nicht ausgegeben wird.

```
LEA.L   PROMPT,A0  Zeiger auf den Prompt
BSR.S   WRSTR      Prompt ausgeben
```

Als nächstes müssen wir den vom Benutzer eingegebenen Text einlesen. Dazu laden wir A3 mit der Adresse eines Puffers für die Zeichen. D2 lassen wir auf das zweite Byte dieses Puffers zeigen, da im ersten Byte die Länge des eingegebenen Textes abgelegt werden soll. Für den Aufruf laden wir D1 mit dem File-Handle für die Standard-Eingabe, den wir in D7 vorliegen haben; D3 enthält die Länge des Puffers. Nun rufen wir die Funktion 'Read' auf, um die Zeichen von der Tastatur einzulesen. Auch diese Funktion terminiert, wenn der Puffer voll ist oder der Benutzer die RETURN-Taste betätigt; sie besorgt auch das Echoen der Zeichen und die Behandlung der Löschtaste. Nach Aufruf dieser Funktion enthält D0 die Anzahl der eingelesenen Zeichen; diesen Wert legen wir in dem durch A3 adressierten Byte ab. Dadurch hat dieser Text das gleiche Format wie unsere anderen Texte.

```
LEA.L   PUFFER,A3  Zeiger auf den Puffer
MOVE.L  D7,D1      stdin-Handle
MOVE.L  A3,D2
ADDQ.L  #1,D2      Zeiger auf den Zeichenbereich
MOVEQ   #30,D3     Länge des Puffers
CALL    Read       Namen einlesen
MOVE.B  D0,(A3)    Länge einfügen
```

Nunmehr können wir die nächsten beiden Texte ausgeben. Der erste ist wieder das altbekannte 'Hallo', der zweite ist der gerade von der Tastatur eingelesene Name.

```
LEA.L   MESS,A0    Zeiger auf den Text
BSR.S   WRSTR      Ausgeben
LEA.L   PUFFER,A0  Zeiger auf den Namen
BSR.S   WRSTR      ... und ausgeben
```

Um das Programm ordnungsgemäß zu beenden, rufen wir die AmigaDOS-Funktion 'Exit' auf. Diese bewirkt eine Terminierung des Programms, wobei der Wert in D1 als Return-Code zurückgegeben wird. Dabei geben wir dem Return-Code einen von Null verschiedenen Wert, wenn wir die AmigaDOS-Bibliothek nicht öffnen konnten.

```
        MOVEQ   #0,D1      Return-Code: OK
EX      CALL    Exit       Terminierung des Programms

FEHLER  MOVEQ   #20,D1     Return-Code: Fehler
        BRA.S   EX
```

Das folgende Unterprogramm dient zur Ausgabe eines Textes. A0 zeigt dabei auf die Länge des Textes, die folgenden Bytes enthalten die einzelnen Zeichen. Der in D6 stehende File-Handle wird für den Aufruf nach D1 geladen. Die Länge des Textes wird in D3 abgelegt, und der entsprechend geänderte Wert von A0 wird als Zeiger auf den Puffer in D2 abgelegt. Anschließend wird die Funktion 'Write' aufgerufen, um die Ausgabe durchzuführen.

```
WRSTR    MOVE.L   D6,D1       stdout-Handle
         MOVEQ    #0,D3       Längenregister löschen
         MOVE.B   (A0)+,D3    D3 := Länge des Textes
         MOVE.L   A0,D2       Text als Schreibpuffer verwenden
         CALL     Write
         RTS
```

Im Datenbereich findet sich der Name der AmigaDOS-Bibliothek und der Platz für den Puffer. Vor den Texten steht in einem Byte deren Länge.

```
         DATA
DOSNAME  DC.B     'dos.library',0
PUFFER   DS.B     1
         DS.B     30
MESS     DC.B     6
```

Glossar

ACIA
Spezialbaustein zum Betrieb einer seriellen Leitung.

Aufwärtskompatibel
Eine Maschine wird als aufwärtskompatibel zu anderen Modellen bezeichnet, wenn für diese geschriebene Programme ohne Änderung auf ihr lauffähig sind.

BCD-Code
Binär codierte Dezimalzahlen. Bei dieser Darstellung von Zahlen wird ein Byte in zwei Nibble aufgeteilt, von denen dann jedes eine dezimale Ziffer darstellt.

Betriebssystem
Programm zur Verwaltung des Computers und seiner Betriebsmittel (Speicher, CPU, Platten, Drucker etc.); kann dem Benutzer das Programmieren erleichtern, indem z.B. Ein- und Ausgabe einfach über Aufrufe von Betriebssystem-Unterprogrammen realisiert werden. Beim gleichzeitigen Ablauf mehrerer Programme notwendig, um diese voreinander zu schützen.

Borrow
'Borgen' beim Subtrahieren; tritt beim Abziehen größerer von kleineren Ziffern auf ('sieben hin, eins im Sinn').

Cache, cachen
Normalerweise holt sich der Prozessor jeden auszuführenden Befehl einzeln aus dem Hauptspeicher. Durch Verwendung von wesentlich schnellerem Cache-Speicher, der direkt auf dem Prozessor-Chip untergebracht ist, ist es möglich, eine Anzahl von Befehlen vor und hinter dem gerade ausgeführten im Prozessor zu speichern. Insbesondere kurze Schleifen werden dadurch sehr schnell ausgeführt, da ihr Code vollständig im Cache-Speicher vorliegt und während der Abarbeitung der Schleife nicht aus dem Hauptspeicher geholt werden muß. Cache-Speicher findet auch zur Zwischenspeicherung von Daten Verwendung.

Co-Prozessor
Spezialprozessor für bestimmte Aufgabenbereiche, beispielsweise Bildverarbeitung oder Gleitkommaarithmetik. Da er im Gegensatz zum Hauptprozessor nicht frei

programmierbar, sondern hardwaremäßig nur für einen eingeschränkten Aufgabenbereich taugt, kann ein Co-Prozessor Aufgaben aus diesem Bereich wesentlich effizienter und damit vor allem schneller bearbeiten.

Debugger

'Entwanzer' von Programmen. Bietet verschiedene Möglichkeiten zur Fehlersuche wie Tracing und Breakpoints.

Displacement

Abstand zwischen dem Anfang eines Datenbereiches und einem speziellen Element innerhalb dieses Bereiches; zeigt ein Register auf den Anfang des Datenbereiches, so kann durch Angabe dieses Registers und des Diplacements auf dieses Element zugegriffen werden.

DMA

direct memory access; direkter Speicherzugriff externer Einheiten ohne Zwischenschaltung des Prozessors. Durch Verwendug von DMA lassen sich wesentliche höhere Datenübertragungsgeschwindigkeiten erreichen, da u.a. durch spezielle DMA-Bausteine die vom Prozessor benötigte Befehlshol- und Befehlsdecodierzyklen entfallen.

Exception

Ausnahmebedingung. Das Eintreten einer Exception veranlaßt den Prozessor, das laufende Programm zu unterbrechen und ein Unterprogramm zur Bearbeitung dieser Bedingung anzuspringen. Sinn einer Exception ist es, einen internen Fehler zu verarbeiten (Trap) oder Kommunikation mit externen Einheiten zu betreiben.

Handler

Exception- oder Interrupt-; Unterprogramm, das vom Prozessor beim Auftreten einer Ausnahmebedingung angesprungen wird.

Interrupt

Von einem peripheren Gerät beantragte Unterbrechung des laufenden Programms; nimmt der Prozessor die Unterbrechung an, so springt er in ein Unterprogramm zur Behandlung der Unterbrechung; Interrupts dienen zur Kommunikation mit dem Prozessor.

LIFO

Last in, first out; bezeichnet die bei einem Stack verwendete Zugriffsmethode: nur der zuletzt abgelegte Wert kann gelesen werden. Eine andere Zugriffsmethode, die gewöhnlich bei Puffern angewandt wird, nennt man FIFO (first in, first out). Beide Zugriffsmethoden dürfen nicht mit der Grundregel der Datenverarbeitung verwechselt werden. Diese lautet **GIGO** (garbage in, garbage out).

Makroprogrammierung

Häufig verwendete Befehlssequenzen, die zu kurz zur Gründung eines eigenen Unterprogramms sind oder außergewöhnlich viele Argumente bearbeiten, lassen sich als Makros schreiben. Dabei wird der Makroaufruf im Programm textuell durch den Makrotext ersetzt, wobei die im Makro verwendeten Variablen durch die im Aufruf spezifizierten aktuellen Parameter textuell substituiert werden.

Memory-mapping

Register von anderen Einheiten, die zur Steuerung peripherer Einheiten dienen, erscheinen dem Prozessor wie normale Speicherstellen. Ein- und Ausgabe werden dabei über bestimmte Adressen vollzogen; bei Memory-mapping entfallen die bei früheren Prozessoren üblichen I/O-Befehle.

Mnemonic

Illustrierende Kurzform eines Wortes; Bsp.: CLR für »clear«, BGT für »branch if greater«.

Monitor

Einfaches Betriebssystem; bietet Möglichkeiten zum Eingeben, Starten und Debuggen von Programmen; stellt mitunter auch einfache I/O-Routinen zur Verfügung.

Nibble

Vier zusammengehörende Bits; ein Nibble bildet eine Ziffer des BCD-Codes.

Offset

siehe Displacement

Patchen

Verändern (meist Anpassen) eines Programms durch direkte Manipulation des Codes. Gepatched wird hauptsächlich dann, wenn das Quellprogramm nicht oder nicht mehr zur Verfügung steht oder der Übersetzungsvorgang unverhältnismäßig aufwendig wäre.

portierbar

Übertragbar; portierbare Programme sind nicht auf einen bestimmten Rechner fixiert, sondern können ohne große Änderungen auf anderen Computern verwendet werden.

Pre-Fetch

Vorgriff auf ein noch nicht benötigtes Wort im Speicher. Während der Abarbeitung eines Befehls holt sich der Prozessor bereits ein oder mehrere Worte, vo denen er annimmt, daß er sie als nächstes benötigt, aus dem Hauptspeicher in seinen eigenen Cache-Speicher. Dies können sowohl Programmbefehle als auch (bei intelligenten Pre-Fetch-Algorithmen) Daten sein.

relokatibel

Verschieblich, kann also an einer beliebigen Adresse im Speicher zum Ablauf gebracht werden.

Save-Area

Datenbereich eines Unterprogramms, in dem dieses bei Einsprung die Inhalte seiner Arbeitsregister rettet. Eine Save-Area kann während des Programmablaufs dynamisch generiert werden, was erst rekursive Programmierung ermöglicht.

Semaphor

Markierung, die vom Prozessor mit einem einzigen, unteilbaren Befehl getestet und gesetzt werden kann. Diese Unteilbarkeit ist notwendig, um mehrere Prozessoren oder Prozeße zusammenarbeiten zu lassen.

Shift, shiften

Verschieben von Bitmustern in einem Register oder Speicherstellen; die dabei herausgeschobenen Bits können verlorengehen oder am anderen Ende wieder nachgeschoben werden; dient u.a. zur Multiplikation und Division mit Zweierpotenzen.

Signiert

Mit Vorzeichen versehen; dieses ist bei Zahlen in Zweierkomplementdarstellung durch das oberste Bits gegeben

Trap

'Falle', in der sich ein Prozessor beim Auftreten von Fehlern (wie etwa bei Division durch Null) selbst fängt; kann auch durch bestimmte Befehle erzeugt werden. Der Prozessor springt ein Unterprogramm zur Behandlung der Fehlerursache an, aus dem er anschließend wieder in das laufende Programm zurückspringen kann.

Index

47
* 41
4040 14
6502 14, 16
6800 14
68000 28
68008 28
68010 28, 127, 136
68020 29, 94, 109, 127, 136
6809 14, 16
80286 15, 19
8080 14, 16
8085 14, 16
8086 15, 16
8088 17
A0-A7 (Adreßregister) 20
Abarbeitung von Exceptions 127
ABCD 104, 106, 107
absolut(e)(r)
  Adressierungsmodus 34, 51, 53
  Code 34
  Sprung 22, 80
  Symbole 34, 38
  Verzweigung 22
ACIA 145, 152, 162, 183
  Auswahl der Charakteristika 66
  Datenport 65, 66
  Interruptmodus 66
  Kontrollport 65, 66
  Polled-Modus 66
  Reset und Initialisierung 69
Ada 87
ADD 91, 92, 93, 97
ADDA 91, 92
ADDI 92
Addition 22, 47, 91
  binär kodierter Dezimalzahlen (ABCD) 104, 106, 107
  binäre (ADD) 91, 92, 93, 97
  extended (ADDX) 92, 93, 99
  mit Direktoperanden (ADDI) 92
  schnelle (ADDQ) 92
  von Adressen (ADDA) 91, 92
ADDQ 47, 92
ADDX 92, 93, 99
Adresse 20, 34, 39
  absolute 34, 39, 51, 53
  effektive 39, 49, 82
  implizite 48
  relative 41, 42
  symbolischer Name 31
Adressierung 20, 34, 39
  indirekt mit Postincrement 45
  indirekt mit Predecrement 45
  indirekt über Adreßregister 43, 45
  Register- 39
  relativ zum Programmzähler 23
Adressierungsarten 39, 48, 49
  alterable 49
  control 49
  data 49
  Direktoperand 47, 51
  memory 49
  Postincrement 45
  Predecrement 45
  Zusammenfassung 48
Adreßfehler 124, 127, 134, 166
Adreßraum 29
Adreßregister (A0 bis A7) 20
aktuelle Position 41
Amiga (Commodore) 188
An (ein beliebiges Adreßregister) 39, 45
AND 106, 112, 116
ANDI 113
ANDI mit CCR 113
ANDI mit SR 113
Anforderung von Speicherplatz 111, 118, 120
Anforderung von Stackbereichen 84
Angehaltener Zustand 133
Anhalten des Prozessors 22
Anweisungen
  Syntax 33, 36, 37
Anwendungen 27
Arithmetik 91
  arithmetische Operationen 22, 111
  arithmetische Operatoren 37
  arithmetischer Links-Shift (ASL) 114
  arithmetischer Rechts-Shift (ASR) 114
  arithmetischer Shift 114
  binäre 104
  dezimale 104, 105, 106, 107, 109
  ganzzahlige 37
  mit mehrfacher Genauigkeit 92
  signierte 95, 100
  unsignierte 95, 100
  überlauf 25, 114
ASL 114
ASR 114

Assembler 31, 32, 34, 37, 41, 53, 93, 139
  Anweisungen 33
  Syntax 31, 36
Assemblersprache 20, 24, 31
Asynchrone Kommunikationsschnittstelle (ACIA) 65
Atari 520ST 186
aufwärtskompatibel 183
Ausdrücke 37, 39
Ausgabe 88, 146, 147
Ausgabe von Hexadezimalzahlen 147
Aussprungfolge 87
Austausch
  von Registerhälften (SWAP) 95, 97
  von Registern (EXG) 95

B.-Kürzel (Byte) 32
Bcc 22, 54, 55, 57, 66
BCC 57
BCD (Binär kodierte Dezimalzahl) 104, 105, 106, 183
BCHG 116, 117
BCLR 116, 117
BCS 57
Bedingter Sprung 22, 53, 55, 57, 62, 80
Befehle
  Monitor- 140
  Speicher- 141
Befehlstabelle 153
Behandlung von Exceptions 128, 141
Behandlung von Traps 141
Benutzer
  Byte im Statusregister 128
  Modus 26
  Registersatz 139
  Stackzeiger (USP) 48, 126, 159
  Status 18, 22, 126, 128
BEQ 54, 57
Betriebssystem 26, 183
BGE 57
BGT 57
BHI 58
BHS 58
Binden (LINK) 25, 29, 87
Binär kodierte Dezimalzahl (BCD) 104, 105, 106, 109
Binäre Arithmetik 104
Bit-Befehle
  Bit testen (BTST) 61, 67, 116
  Bit testen und löschen (BCLR) 116, 117
  Bit testen und modifizieren (BCHG) 116, 117
  Bit testen und setzen (BSET) 116, 117
  Test-and-Set (TAS) 18, 116, 117
BLE 58
BLO 58
BLS 58
Borrow 183
BPL 58
BLT 57
BMI 57
BNE 54, 55, 57, 67
BRA 57, 75, 80, 82, 84
Breakpoint 141, 158, 159, 163, 164, 167, 168
  löschen 139
  setzen 139
BSET 116, 117
BSR 57, 76, 78, 82, 84, 87, 147, 161
BTST 61, 67, 116
Bus 27
  Steuerungslogik 27
Busfehler 74, 124, 127, 134, 136, 166
  doppelter 136
BVC 57
BVS 57
Byte (8 Bit) 20

C (Bit im Statusregister) 22
Cache 183
Carry-Bit (übertrag-Bit) 22
CCR (Condition-Code-Register) 113
CHK (Test mit eventuellem Trap) 24, 25, 29, 132
CLR 29, 33, 60, 61, 64
CMP 22, 55, 56, 60
CMPA 56
CMPI 56
CMPM 56, 107
Co-Prozessor 29, 184
COBOL 104
Code
  absoluter 34
  positionsunabhängiger 22, 34, 35, 41, 72, 81, 139
  re-entrant 41
  reiner 41, 72
  seriell wiederverwendbarer 41
Commodore Amiga 188
Compiler 25, 93
Condition-Codes
  Erweiterungsflag (X) 22
  invertieren 113
  löschen 113
  Negativflag (N) 22
  Nullflag (Z) 22
  Register (CCR) 113
  setzen 22, 55, 57, 62, 64, 113
  testen (Bcc, DBcc, Scc) 22
  Überlauf (V) 22, 114
  Übertrag (C) 22

Daten
  Direkt- 47, 54, 60, 61
  Operanden 49
  PC-relative Adressierung 23
  Register (D1 bis D7) 20
  Speicherung 118
  Übertragung zu Peripheriegeräten 26
Datenport des ACIA 65, 66
DBcc 58, 63, 64, 89
DBEQ 65
DBF 63
DBRA 63, 64, 66
DC-Anweisung 36
Debugger 24, 134, 139, 141, 184
Define Constant (DC) 36
Define Storage (DS) 35
Dekrementieren, Test und Sprung

bedingt (DBcc) 22, 58, 63, 64, 89
if equal (DBEQ) 65
if false (DBF) 63
unbedingt (DBRA) 63, 64, 66
Dezimale Arithmetik 104, 105, 106, 107, 109
Direkte
Addition (ADDI) 92
Adressierungsart 47, 51
AND (ANDI) 113
Daten 47, 54, 60, 61
EOR (EORI) 113
OR (ORI) 113
Subtraktion (SUBI) 93
Vergleich (CMPI) 56
Direkter Speicherzugriff (DMA) 27, 133
Direktoperand 47,51
Displacement 184
Division 29, 100, 101, 114
lange 101
Null-Trap 101
signierte (DIVS) 100, 132
unsignierte (DIVU) 100, 101, 132
Division von Vollworten 101
DIVS 100, 132
DIVU 100, 101, 132
DMA (Direkter Speicherzugriff) 27, 133, 184
Dn (beliebiges Datenregister) 39
DS-Anweisung 35

Ein-/Ausgabe 65
Ein-/Ausgabebereich 60
Ein-Chip-Computer 15
Einfache Monitorbefehle 153
Einfaches Speicherdiagnoseprogramm 58
Einführung in den 68000 20
Eingabe 149
Eingabe eines Anwenderprogramms im Monitor 158
Einsprungfolge 86
Einlesen einer hexadezimalen Zahl von der Tastatur 149
Einlesen eines Zeichens 144
Einstellung des ACIA 65
END-Anweisung 36
Entbinden (UNLNK) 25, 29, 87
EOR 112
EOR mit CCR 113
EOR mit SR 113
EORI 113
EPROM 81
EQU-Anweisung 34, 58
Exception-Handler 127, 128, 159
Exception-Vektor 123, 124, 127
Exception 47, 123, 126, 141, 184
Adreßfehler 127
Busfehler 127
externe 123
interne 123
Reset 127
Routinen zur Behandlung 127, 128, 141
EXG 95
Exklusives Oder (EOR) 112
EXT 95, 96
Externer Reset 124

Fehlermeldungen 23
FIFO 185
FORTRAN 24
Freigabe von Speicherbereichen 118

Ganzzahlarithmetik 37
Geschichte der Mikroprozessoren 14
GIGO 185
Großrechner 15

Handler 184
Hardware-Traps 23
Hauptspeicher 20
Hexadezimal 34, 116, 140
Home-Computer 14

iAPX 432 15, 19
Illegale Operationen 23, 124
Illegaler Befehl 130
Exception durch 124
Implizite Adressierung 48
Indexregister 42, 43
Indirekte Adressierung 43
Initialisierung der Behandlung von Kommandos 151
Integrierte Schaltkreise 14
Intel 4040 14
Intel 80286 15, 19
Intel 8080 14, 16
Intel 8085 14, 16
Intel 8086 15, 16
Intel 8088 17
Intel iAPX 432 15, 19
Interne Exceptions 123
Interne Register 20
Interrupt 26, 46, 123, 126, 129, 165, 184
Ebene 18
Maske 127
Modus des ACIA 66
Prioritätsmaske 22
Routinen 117, 144
Vektoren 27
Invertieren des Condition-Codes 113
Invertieren des Status 113

JMP 33, 35, 41, 48, 80, 81
JSR 82, 87, 24

Klammern 43
Kleinbuchstaben 151
Kommentare 32, 36
Kontrollport des ACIA 65, 66
Kürzel (B, W, L) 32, 42, 45

L.-Kürzel (Vollwort) 32, 42, 45
Label (Marke) 33, 41
Laden eines Programms 22
LEA 83
LIFO 185

LINK 25, 29, 87
Linksrotation (ROL) 115
Linksshift
arithmetischer (ASL) 114
logischer (LSL) 114
Logische Operationen 22, 111, 116
Exklusives Oder (EOR) 112
Komplement (NOT) 111
Linksshift (LSL) 113, 114
Oder (OR) 106, 111, 112
Rechtsshift (LSR) 109, 114
Shift 113, 114
Und (AND) 106, 112, 116
LSL 113, 114
LSR 109, 113, 114
Längenkürzel 32, 42, 45
Löschen
Breakpoints 139
Cond 113
Speicherstellen (CLR) 29, 33, 60, 61, 64
Status 113
Überlauf-Bit 114

Makroprogrammierung 185
Marke 33, 41
Maschinencode 25, 26, 31
Maske 127
Maskierung 112, 116
Mehrfache Exceptions 136
Mehrfache Rechengenauigkeit 92
Memory-mapping 185
Minicomputer 15
Mnemonics 31, 185
Modifikation von Benutzerregistern 139
Modifikation von Registern 140
Modifikation von Speicherstellen 139
Modus
Benutzer 26, 139
Prozessor 26
Supervisor 139
Trace 22, 24, 139
Monitor 26, 139, 140, 158, 185
MOS Technology 6502 14, 16
Motorola 6800 14
Motorola 68000 15, 28
Motorola 68008 28
Motorola 68010 28, 127, 136
Motorola 68020 29, 94, 109, 127, 136
Motorola 6809 14, 16
MOVE 51, 52, 53, 54, 55, 57, 60, 61, 68, 69, 72, 74, 82, 83, 87, 92, 97, 117, 126
MOVE SR 29, 128
MOVE nach CCR 128
MOVE mit USP 126, 133
MOVEA 53, 82
MOVEC 28
MOVEF 29
MOVEM 25, 26, 73, 74, 75, 79, 87, 128
MOVEP 26, 68, 69
MOVEQ 47, 61
MOVES 28
MULS 94, 164
Multiplikation 29, 94, 96, 114
signierte (MULS) 94, 164
unsignierte (MULU) 94, 97
von Vollworten 96
Multiprozessorsystem 15
MULU 94, 97

N (Negativflag im Condition-Code) 22
National Semiconductor NS 16032 15, 19
NBCD 104, 108
Negation 94
binär extended (NEGX) 94, 108
binär kodierter Dezimalzahlen (NBCD) 104, 108
binäre (NEG) 94
NEG 94
NEGX 94, 108
Nibble 116, 185
Nichtimplementierter Befehl 124, 130
NMI (Nichtmaskierbarer Interrupt) 129
NOP (keine Operation) 31
NOT 111, 112

Offset 185
Operand
control 49
data 49
memory 49
Quell- 32
Reihenfolge im Assembler 32
Wort- 21
Ziel- 32
Operandenadresse 39, 49, 82
Operationen
arithmetische 22, 111
auf einzelnen Bits 116
logische 22, 111, 116
OR 111, 112
ORG-Anweisung 34, 40
ORI 113
ORI mit CCR 113
ORI mit SR 113

PACK 29, 109
Parität 66
Pascal 24, 87
Patchen 185
PC (Programmzähler) 41
PEA 84, 162
Periphere Geräte 26
Physikalische Register 126
Polled-Modus des ACIA 66
portierbar 186
Portabilität 24
Positionsunabhängiger Code 22, 34, 35, 41, 72, 81, 139, 152
Postincrement 64, 71, 72, 74, 87
Predecrement 54, 71, 72, 74, 87, 108
Prefetch 74, 186
Prioritätsebene 127
Prioritätsfolge von Exceptions 137
Privileg 26
Ebene 126
Verletzung 132

Privilegierte Befehle im Benutzermodus 124
Programmzähler (PC) 21, 41
Puffer, zyklischer 144

RAM 15, 17
read-modify-write 27, 117
reentrant 41
Register 20, 28
  anzeigen und verändern 154
  direkte Andressierung 39
  Gebrauch von Mnemonics 31
Reihenfolge der Operanden in Assemblerzeilen 32
Reiner Code 41, 72
relative
  Adressierung 23, 41, 42
  Angabe des Sprungziels 23
relokatibel 186
relokatible
  Abschnitt 42
  Programme 34
  Symbole 34, 38, 42, 43
  Werte 35
Relokationsinformation 35
Reservierung von Speicherbereichen 118
RESET 126, 133
Reset Exception 127, 130
Reset und Initialisierung des ACIA 66
Reset-Vektor 125
Retten von Werten aud dem Stack 72
Return and Store (RTR) 128
Return from Exception (RTE) 126, 128
Röhre 14
ROL 115
ROM 15, 17, 152
ROR 109, 115
RORG-Anweisung 34
Rotieren 113, 115
  links (ROL) 115
  links extended (ROXL) 115
  rechts (ROR) 109, 115
  rechts extended (ROXR) 115
ROXL 115
ROXR 115
RTE 127, 128
RTR 128
RTS 76, 79, 84, 128, 147, 150, 162
Rücksetzen (resetten) 66, 125, 130, 151
Rücksprung aus einem Unterprogramm (RTS) 76, 79, 84, 128, 147, 150, 162
Rücksprungadresse 76
Rückverweise 40
Rückwärtssprung 80
Rückwärtsverzweigung 55

Save-Area 142, 186
SBCD 104, 108
Scc 22, 29, 58, 62, 63
Schleife 54, 63, 64
Schnelle Form von Befehlen 47
  Addition (ADDQ) 47, 92
  Subtraktion (SUBQ) 47
  Transport (MOVEQ) 47, 61
Semaphor 117, 186
Setzen
  auf Null (CLR) 33, 60, 61, 44
  Breakpoints 139
  Condition-Code 22, 55, 58, 62, 64, 113
  Status 113
  Überlaufflag 114
Shift 69, 113, 114, 186
  arithmetisch 114
  arithmetisch links (ASL) 114
  arithmetisch rechts (ASR) 114
  links 114
  logisch 113, 114
  logisch links (LSL) 113, 114
  logisch rechts (LSR) 114
  rechts 114
  zyklisch 115
Sign-Extend (EXT) 95, 96
Sign-Extension 40
signiert 186
Signierte Arithmetik 95, 100
Signierte Division 100, 132
Sinclair QL 183
SP (Stackzeiger) 20, 48, 71, 72, 73, 75, 77, 79, 87, 126
Speicher
  Adressierungsarten 49
  Anforderung 111, 118, 120
  Ausgabe- und Modifikationsroutinen im Monitor 160, 161
  Befehle im Monitor 141
  Diagnoseprogramm 58
  Freigabe 118
  Kontrolle 117
  Operanden 49
  Reservierung 118
  Routine zur Feststellung der Größe 137
  Speicherstellen untersuchen und verändern 139
  Test 88, 89
  virtueller 28
Spezifikationen von Prozessoren 15, 16, 17, 18, 20
Spitze des Stacks 71
Sprung 21, 22, 40, 43, 53, 75, 80, 128, 147
  absoluter 22, 80
  an eine feste Adresse beim Debugging 24
  Befehl (JMP) 33, 35, 41, 48, 80, 81
  in ein Unterprogramm (JSR) 82, 87
  relativer 23
  vorwärts 80
  rückwärts 80
Sprungtabelle 150
SR (Statusregister) 21, 48, 113, 123, 126, 127, 128
SSP (Supervisor-Stackzeiger) 48, 75, 126, 127
Stack 25, 45, 48, 71, 72, 73, 74, 75, 76, 79, 82, 84, 85, 86, 87, 106, 127, 128, 147, 162, 166
  Benutzerzeiger (USP) 18, 48, 126, 127, 159
  Zeiger (SP) 20, 48, 71, 72, 73, 74, 75, 77, 79, 87, 126
  Anwender- 128
  Speicherzuweisung 84
  System- 127, 168
  Spitze 71
Status
  -Register (SR) 21, 48, 113, 123, 126, 127, 128
  invertieren 113
  löschen 113

setzen 113
Steuerung von Peripheriegeräten 22, 26
STOP 126, 133, 136
SUB 93, 107
SUBA 93
SUBI 93
SUBQ 47, 93
Subtraktion 37, 47, 93
Adressen (SUBA) 93
binär (SUB) 93, 107
binäre kodierte Dezimalzahlen (SBCD) 104, 108
Direktoperanden (SUBI) 93
extended (SUBX) 93
schnell (SUBQ) 47, 93
SUBX 93
Supervisor 126
Bit 22
Modus 15, 26, 139
Stack 127, 128
Stackzeiger (SSP) 48, 126, 127
Status 18, 20, 22, 28, 126, 127, 128
SWAP 95, 97
Symbol (absolut) 34, 37
Symbol (relokatibel) 34, 38, 42, 43
Symbolischer Name 33
Syntax von Befehlen 37
Systemregister 124
Systemstack 127, 168

TAS 27, 116, 117
Test
and Set (TAS) 18, 27, 116, 117
auf Null (TST) 61
Bits (BTST) 61, 67, 116
Condition-Codes 22, 53, 62, 63, 64
TMS9900 17
TMS99000 15, 17
TMS9940 14, 17
TMS9980 17
TMS9981 17
TMS9985 17
TMS9995 17
Trace 113, 124, 127, 134, 158
Bit 22
Exception 168
Modus 22, 24, 139
Transistor 14
Transportieren
Adressen (MOVEA) 53, 82
Daten (MOVE) 51, 52, 53, 54, 55, 57, 60, 61, 67, 69, 72, 74, 82, 83, 87, 92, 97, 117, 126
kleine Zahlen 61
mehrfach (MOVEM) 25, 73, 74, 75, 79, 87, 128
schnell (MOVEQ) 61
USP 106, 126
vom SR 128
vom und zum Stack (MOVEM) 25, 26, 73, 74, 75, 79, 87, 128
zum CCR 128
zum SR 128
zur Peripherie (MOVEP) 26, 68, 69
TRAP 24, 26, 130, 134, 168
Trap 123, 126, 141, 158, 165, 186
-Exception 24
-Vektor 126, 139
bei überlauf (TRAPV) 24, 25, 101, 132
Division durch Null 101
Hardware- 23
verursachen (Debugger) 24
TRAPV 24, 101, 132
TST 60, 61, 67

Überlauf 114
mehrfach (MOVEM) 25
Übertrag-Bit 22
Übertragung von Daten zu Peripheriegeräten 26
Umwandlung
in Kleinbuchstaben 151
von Hexadezimalzahlen in Zeichen 116
UNLK 87
Unsignierte Arithmetik 95, 100
Unsignierte Division 100
Unterprogramm 74, 75, 76, 78, 79, 80, 82, 84, 85, 86, 87, 88, 96, 101, 105, 109, 123, 128, 144, 146, 153
Aufruf 46
Binden (LINK) 86, 87
Entbinden (UNLNK) 87
Sammlung 78
Verschachtelung 79
Unterstützung höherer Programmiersprache 24
Untersuchung der Registersatzes des Benutzers im Monitor 139
Untersuchung von Speicherstellen im Monitor 139, 140
USP (Benutzer-Stackzeiger) 48, 75, 126, 159

V (Überlaufflag) 22, 114
Vektor-Basisregister 28
Vergleichen (CMP) 22, 55, 56, 60
Adressen (CMPA) 56
Direktoperanden (CMPI) 56
mit Null (TST) 61
Speicherstellen (CMPM) 56, 108
Verschachtelung von Unterprogrammen 79
Verweis
rückwärts 40
vorwärts 40
Verzweigung 21, 22, 147
absolute 22
bedingte (Bcc) 22, 54, 55, 57, 62
if carry clear (BCC) 57
if carry set (BCS) 57
if equal to zero (BEQ) 54, 57
if greater than (BGT) 57
if greater than or equal (BGE) 57
if high (BHI) 58
if high or the same (BHS) 58
if less than (BLT) 57
if less than or equal (BLE) 58
if low (BLO) 58
if low or the same (BLS) 58
if minus (BMI) 57
if not equal to zero (BNE) 54, 55, 57, 67

if overflow clear (BVC) 57
if overflow set (BVS) 57
if plus (BLP) 57
ins Unterprogramm (BSR)
57, 76, 78, 79, 82, 84, 87, 147, 161
Rückwärts- 55
unbedingte (BRA) 57, 75, 80, 82, 84
Virtueller Speicher 28
Vollwort (32 Bit) 20
Vorwärtssprung 80
Vorwärtsverweise 40

W.-Kürzel 32, 42, 45
Warmstart 81
Wort (16 Bit) 20

X (Extendedflag) 22

Z (Nullflag) 22
Z80 14, 16
Z8000 15, 18
Zeichen ausgeben 146
Zeichenkette 36
Zeichenkonstante 36, 37
Zilog 14, 15, 16, 18
Zusammenfassung der Adressierungsarten 48
Zuweisung freien Speichers 118
zyklischer Shift 115

T. King & B. Knight

Programmierung des M68000

Für alle Programmierer, die leistungsstark programmieren wollen. Für alle, die den Prozessor ihres Macintosh, Sinclair QL, Atari 520 ST (Jackintosh), Commodore Amiga und weiterer Computer näher kennenlernen wollen.

Dieses Buch behandelt die Programmierung des M68000, des wohl am weitesten entwickelten der weltweit zur Zeit verfügbaren Mikroprozessoren. Es ist ein umfassendes, sich auf den Mikroprozessor aus der Sicht des Programmierers konzentrierendes Handbuch.

Die Grundideen der Prozessorarchitektur des M68000 werden ebenso erklärt wie jeder einzelne seiner Befehle, mit all seinen Fußangeln und möglichen Anwendungen. Der Leser wird ermuntert, effiziente, sichere und transparente Programme zu entwickeln.

Unter den vielen angegebenen Beispielen sind u.a. Programmstücke zur Fehlersuche, Interrupthandling und Speicherzuweisung. All diese Beispiele sind praktisch erprobt, ausführlich beschrieben und dokumentiert und können somit ohne weiteres als Programmteile in neuen Entwicklungen Verwendung finden. Des weiteren wird ein kleines Monitorprogramm entwickelt, mit dessen Hilfe sowohl Ein- und Ausgabe gehandhabt werden als auch Testprogramme zum Ablauf gebracht werden können.

Programmierung des M68000 wird Anfängern wie Fortgeschrittenen eine wertvolle Hilfe bei der Programmierung dieses Mikroprozessors sein. Es ist klar und jargonfrei geschrieben; Erfahrungen mit Assemblerprogrammierung werden nicht benötigt, jedoch sollte der Leser mit den einfachsten Begriffen der Computerterminologie vertraut sein. Schwierige und bei bisherigen Mikroprozessoren kaum verwendete Ausdrücke werden in einem Glossar behandelt.

ADDISON-WESLEY VERLAG (DEUTSCHLAND) GmbH

ISBN 3-925118-07-1

VVA Nr. 563-00007-8